Dicembre 1997

F. Spagna

IL LINGUAGGIO JAVA COME PROPOSTA AVANZATA PER LE TECNOLOGIE INFORMATICHE DELLA DIVISIONE ERG-SIEC

F. Spagna

IL LINGUAGGIO JAVA COME PROPOSTA MODERNA PER LE TECNOLOGIE INFORMATICHE DELLA DIVISIONE ERG-SIEC

1. Presentazione e generalità del linguaggio Java

1.1 Introduzione

Il linguaggio Java[1] è un linguaggio di programmazione orientato agli oggetti presentato pubblicamente da Sun Microsystems nel maggio del 1995, dopo circa quattro anni di sviluppo. All'origine (il suo primo nome è stato *Oak*) esso era stato concepito come un linguaggio per l'elettronica di consumo intelligente (*Green Project*), in seguito orientato ai *set-top box* per la televisione interattiva, con scopi inizialmente diversi da quello per cui successivamente è stato maggiormente utilizzato, cioè la rete Internet. Il suo ideatore, James Gosling, che aveva all'inizio tentato di estendere il linguaggio C++ agli scopi prefissati, decise infine di creare un linguaggio nuovo, abbandonando il C++ che si andava dimostrando poco adatto alle richieste, rinunziando alla compatibilità con esso, pur conservandone in larga misura i costrutti e la sintassi (ma deve essere detto che concettualmente, a parte questi aspetti pur importanti ma più formali, nella sostanza Java assomiglia molto di più al linguaggio Smalltalk che al C++). E' stato solo in un secondo tempo che le sue stesse caratteristiche che gli erano state richieste all'inizio, che erano principalmente rivolte a semplicità, compattezza, velocità, efficienza, ma soprattutto portabilità su dispositivi hardware diversi (con indipendenza dal processore e dal sistema operativo), sicurezza e affidabilità, gli sono valsi la sua elezione a linguaggio ideale per programmi distribuiti sulla rete Internet (chiamata anche *World Wide Web* o più semplicemente *Web*), alla quale sono in generale collegati i più diversi sistemi, e il progetto fu riorientato verso il Web.

JAVA [1] Il nome Java è stato preso da quello usato nel gergo americano per indicare il caffè, e questo spiega il largo uso di simboli e nomi relativi al caffè in riferimento a Java, come per esempio il logo del linguaggio rappresentato da una tazza di caffè fumante riprodotto qui accanto, o, tanto per fare un altro esempio, il termine di Java Beans (che significa grani di caffè) adottato per i componenti software basati su Java.

Dopo la sua presentazione Java ha presto suscitato un grande interesse nel mondo Internet per le nuove possibilità interattive e multimediali che veniva ad offrire alla rete. Le capacità di **interazione** tra l'utilizzatore (*client*) di un browser (programma di navigazione Web) ed un server Web permesse da Java con le applet, che sono applicazioni Java inserite in una pagina HTML (***HyperText Markup Language***) trasmesse in rete da un server Web ed eseguite sulla macchina locale del visitatore Web (*client*), sono molto ampie e vanno ben oltre l'interattività permessa fino ad allora dall'HTML, che si limitava alla semplice lettura della pagina statica con la sola possibilità di selezione di link o campo di un'immagine (*image map*), o al massimo di riempimento di campi di testo in un modulo (*form*) il cui contenuto è inviato indietro al server, il quale esegue in relazione ai parametri ricevuti dei programmi in modalità CGI (*Common Gateway Interface*) e ne rispedisce i risultati al client. Con le applet Java una parte dell'attività viene invece spostata dal server al client e ciò comporta un più alto livello di ***interattività***, un ***alleggerimento dell'attività del server*** ed una ***maggiore sicurezza*** in quanto la tecnica CGI, che aveva costituito prima dell'avvento di Java la maggior parte dell'interattività client/server in rete, ha sempre rappresentato un punto debole dal punto di vista della sicurezza per i server Web, e, trasferendo buona parte del processamento dei *form* a livello del client, ne vengono ridotti i rischi. Con Java si sono viste aprire inoltre nuove possibilità **multimediali** (animazioni, suoni, video e giochi) per la rete Internet.

Ma, pur essendo stato quello delle applet l'uso prevalente di Java all'inizio della sua diffusione, c'è da dire che Java non si limita ad apportare, con le applet, maggiori potenzialità alle pagine Web, perchè esso è un vero e proprio **linguaggio di programmazione** di terza generazione (3GL) a tutti gli effetti, di uso generale e per di più molto avanzato, con il quale si possono creare anche applicazioni indipendenti e non legate ad un browser e con cui si può fare tutto quello che si fa con altri linguaggi di terza generazione (come il C++ o il Fortran), e anzi, rispetto ai linguaggi che lo hanno preceduto, esso ha potuto beneficiare in partenza dell'esperienza già acquisita su quelli, che ha permesso miglioramenti in particolare dal punto di vista della semplicità e della sicurezza, ma in più è stato arricchito da capacità avanzate di funzionamento **multipiattaforma**, di sviluppo di **applicazioni distribuite** e dotato di funzionalità intrinseche di **multithreading**.

Così, dopo le applet, Java ha visto il suo uso estendersi alle applicazioni server e poi nei settori più vari. Java viene proposto da Sun non solo come un linguaggio, ma anche come una piattaforma.

1.2 Applicazioni Java e Java Virtual Machine

Il linguaggio di programmazione Java permette di creare sia delle ***applet*** funzionanti su pagine HTML ed eseguite da un browser, sia delle ***applicazioni*** Java a sè stanti (*standalone*). Le **applet** sono delle applicazioni (come indica il loro stesso nome, che in inglese sta per "piccola applicazione", perchè inizialmente furono adoperate solo applicazioni di dimensioni ridotte per via della larghezza di banda permessa dalla rete, ma nulla impone in linea di principio delle limitazioni alle dimensioni delle applet), che, richiamate in un documento HTML in modo analogo a quanto viene fatto ad esempio per le immagini (in questo caso però con il *tag* specifico `<applet>` anzichè quello `<img>`), sono trasmesse sulla rete dal server e scaricate (*downloaded*) sulla macchina client, sulla quale sono eseguite e visualizzate tramite l'interfaccia grafica del browser in un riquadro predisposto per esse sulla pagina HTML in cui sono richiamate analogamente a quanto si fa per le immagini, purchè il browser utilizzato sia dotato di un suo interprete Java. Esse possono essere scritte in modo da rispondere interattivamente ad azioni del visitatore Web (mouse, tastiera, etc.). Le **applicazioni Java** sono invece delle vere e proprie applicazioni indipendenti, che poggiano direttamente sul sistema operativo senza l'intermediazione di un browser, pur richiedendo comunque sempre un interprete per l'esecuzione.

La rilevanza data all'uso di applet nei primi tempi dell'apparizione di Java sull'Internet, che è prevalso nella prima utilizzazione di questo linguaggio nel Web, anche perché ne costituiva la parte più visibile ed innovativa, non deve far pensare che le applet siano le sole applicazioni Java di rete significative, perché l'uso ad esempio di programmi Java funzionanti sul lato server è oggi almeno altrettanto importante.

In ogni caso il codice sorgente Java, scritto in uno o più file di testo, caratterizzati dall'estensione `.java`, viene compilato con il compilatore Java, che è un'appliczione chiamata *javac* e scritto esso stesso in linguaggio Java, presente nel **Java Developer's Kit** (**JDK**, ambiente di sviluppo Java della Sun di cui esistono versioni per diverse piattaforme), che ne ricava un cosiddetto *bytecode* sotto forma di un file binario con estensione `.class`) distinto per ogni classe.

Il **bytecode** è un codice intermedio costituito da una sequenza di istruzioni macchina (*p-code*), analogo ad un codice binario eseguibile, ma compilato per una macchina virtuale intermedia universale, invece che per una specifica architettura di computer e sistema operativo (qualcuno [1.1] parla di un codice compilato all'80%). Le applicazioni così compilate restano in tal modo indipendenti dalle particolarità di sistema operativo e hardware specifici della macchina su cui devono girare e possono essere eseguite su qualunque sistema. Ciò è fondamentale per le applet che, come le pagine HTML su cui esse sono presentate, devono essere viste su qualunque piattaforma di sistema collegato alla rete che le utilizzi, ma anche per le applicazioni Java vere e proprie che possono con grande vantaggio essere scritte e sviluppate una volta per tutte (*"write once, run anywhere"* è il motto della Sun riferito a Java).

Per la portabilità su ogni piattaforma si potrebbe anche teoricamente pensare di inviare sulla rete un codice sorgente non ancora compilato, ma il vantaggio della trasmissione invece sulla rete di un bytecode già portato ad un certo livello di compilazione è costituito dalla sua maggiore compattezza, sicurezza ed efficienza di esecuzione.

Non è esclusa la possibilità di produrre bytecode Java servendosi di un linguaggio diverso da Java: un interessante soluzione di questo genere è stata realizzata da Aonix con l'applicativo *Object Ada*, che genera bytecode Java a partire da un programma in linguaggio Ada [#], e anche con il *Delphi* di Inprise, che usa come linguaggio l'Object Pascal, è oggi possibile produrre bytecode Java.

A livello dell'esecuzione del bytecode è necessario un **ambiente di run-time** di Java rappresentato da un **interprete**, detto anche **Java Virtual Machine** (macchina virtuale, spesso chiamato brevemente **JVM** o anche semplicemente **VM**), proprio della piattaforma (cioè dal tipo di processore e di sistema operativo) sulla quale avviene l'esecuzione (ciascuna piattaforma deve infatti avere il suo particolare interprete Java), che interpreta il bytecode, producendo a partire da esso il necessario ***codice macchina nativo*** finale, questa volta specifico della piattaforma di esecuzione, solo al momento in cui l'applicazione viene eseguita e cioè al *run-time*. In tal modo uno stesso bytecode è eseguibile come tale su qualunque macchina sia dotata del suo proprio particolare interprete Java. Il passaggio attraverso questo processo di interpretazione all'atto dell'esecuzione penalizza però dal punto di vista della velocità di esecuzione rispetto ad un codice che fosse stato invece già in precedenza totalmente compilato e che cioè interagisse direttamente con il sistema operativo e l'hardware, anche se si può riconoscere che la fase finale di interpretazione risulta comunque molto semplificata grazie alla precedente fase di compilazione a livello di bytecode. E questo è il prezzo che si viene a pagare come corrispettivo del grande vantaggio che viene offerto dall'opportunità di un possibile unico sviluppo delle applicazioni indipendente dalla particolare piattaforma su cui l'applicazione deve essere eseguita. Tale indipendenza dalla piattaforma è poi evidentemente addirittura essenziale per applicazioni che devono funzionare su una rete eterogenea quale è il Web. C'è da aggiungere infine che il passaggio attraverso il bytecode presenta anche il vantaggio di permettere l'effettuazione di vari controlli di sicurezza al momento dell'esecuzione (vedi paragrafo 1.6).

I browser abilitati a far funzionare le applet dispongono di un interprete Java integrato in essi, ma il JDK di Sun comprende anche un'applicazione speciale chiamata **appletviewer** che permette di eseguire le applet fuori da un browser.

Esiste la possibilità di unire ad un codice Java anche del codice nativo (per esempio in linguaggio C) proprio di una macchina specifica, ma in tal modo si vengono a perdere le caratteristiche di portabilità del linguaggio.

Per risolvere il problema dello scadimento delle prestazioni conseguente all'interprete, molte delle ultime versioni di Virtual Machine per varie piattaforme adottano un sistema di compilazione, detta **compilazione *just-in-time*** (comunemente abbreviato in **JIT**), che noi diremmo "al volo", per la conversione del bytecode Java in codice di macchina nativo specifico del processore in uso, eseguita automaticamente al momento del suo caricamento (della sua esecuzione): le prestazioni possono migliorare in tal modo di molto, arrivando a diventare addirittura paragonabili a quelle di un codice che fosse stato compilato direttamente in forma nativa (ad esempio applicazioni compilate in C++).

Una soluzione ancora più avanzata è quella di una compilazione dinamica con la quale diverse sezioni del programma sono compilate con diversi livelli di ottimizzazione.

L'interprete di Java richiede poca memoria RAM: circa 40 kB da solo, cui si devono aggiungere circa 175 kB per il multithreading e le librerie standard.

Da un programma scritto in Java può anche essere creato un eseguibile indipendente che può girare senza bisogno del supporto dell'ambiente di runtime Java, esattamente come se fosse un programma compilato in C o C++: per questo c'è bisogno di un compilatore che converte il codice sorgente in codice di macchina binario nativo (per esempio il Jikes di AlphaWorks). Naturalmente un tale programma perde ogni portabilità perché può essere eseguito solo sulla piattaforma per la quale è stato compilato.

1.3 Java Virtual Machine e prestazioni di Java

Oltre a quella compresa nel JDK 1.1 di SunSoft, esistono anche altre implementazioni di Virtual Machine di Java, rispondenti sempre alle specifiche di Sun, ma prodotte da sviluppatori diversi. Per la piattaforma Windows ci sono in particolare quella di Microsoft (usata nel browser Internet Explorer), quella di Symantec e quella integrata nel browser Web di Netscape. Le ultime versioni di VM comprendono tutte un compilatore *just-in-time (Jit)*. Però le diverse Virtual Machine oggi disponibili purtroppo presentano ancora comportamenti abbastanza diversi sulle diverse piattaforme, lasciando ancora aperti dei problemi di portabilità per gli sviluppatori che ne devono tenere conto.

Il nuovo **Java Performance Windows Runtime** (**JPWR**)# di Sun comprende una VM per Windows con un compilatore (di Symantec) particolarmente efficiente. Con la versione 1.2 del JDK è stato introdotto il compilatore JIT **HotSpot** (sviluppato da Longview Technologies), che porta a prestazioni addirittura del livello di quelle ottenibili con compilatori C/C++, fornito anche di un nuovo meccanismo di *garbage collection* non distruttivo chiamato *Train*. Anche IBM sta lavorando su quella che chiama **Universal Virtual Machine** per Java da cui si attendono prestazioni molto elevate.

[1.x] PCWeek 1937/n.92, 6 ottobre 1997, pag. 65.

1.4 Java Developer's Kit

Il **Java Developer's Kit** (abbreviato generalmente in **JDK**) di Javasoft, che è una divisione di Sun, include una serie di librerie di classi per l'interfaccia utente, l'input/output, funzionalità di rete con protocolli Internet. Tali librerie sono scritte in Java e quindi totalmente portabili.

Microsoft ha dal canto suo sviluppato il **Software Development Kit for Java** (**SDKJ**), attualmente in versione 3.1#, di cui la società vanta le prestazioni, che supporta il JDK 1.1 (ad eccezione del RMI e del JNI), l'integrazione bidirezionale tra gli ActiveX ed i JavaBeans, il J/Direct (classi per l'accesso diretto alle API Win32 di Windows) e una sicurezza basata su permessi. Esso contiene le **Application Foundation Classes** (**AFC**), insieme di classi per componenti di interfaccia grafica ed effetti grafici, il compilatore *jvc* ed un applicazione *awt2afc* per la trasformazione di applet da un'interfaccia grafica di tipo AWT ad una di tipo AFC. Però Sun contesta l'SDK come non del tutto compatibile con le specifiche del suo linguaggio Java dato in licenza a Microsoft, in quanto esso conterrebbe alterazioni delle classi dei package standard (core) `java.lang`, `java.awt` e `java.io` e in più non sopporta il JNI e l'RMI (si vedrà più tardi che cosa sono questi). In seguito a questa disputa si teme che Microsoft possa arrivare a creare un suo standard di Java non conforme a quello di Sun minacciando l'universalità di Java.

1.5 Caratteristiche del linguaggio

La Sun, nel presentare il Java, lo ha definito come un linguaggio avente le seguenti caratteristiche: interpretato, neutro rispetto all'architettura, orientato agli oggetti, semplice, distribuito, sicuro, robusto, portabile, ad alte prestazioni, dinamico, e dotato di capacità di multithreading. Passiamo in rassegna questi vari aspetti.

Sul fatto che il Java sia **interpretato** e **neutro rispetto all'architettura** si è già parlato al paragrafo 1.2.

Il linguaggio Java è un **linguaggio orientato agli oggetti** (la programmazione ad oggetti è detta in breve anche OOP, da *Object-Oriented Programming*) la cui sintassi e struttura ad oggetti sono derivate direttamente dal C++, e ciò costituisce certamente un grande vantaggio dal punto di vista dell'apprendimento e della conoscenza, essendo il C++ di gran lunga il più diffuso ed usato tra i linguaggi di programmazione orientati agli oggetti. Ma va detto che Java recepisce diversi concetti e caratteristiche interessanti anche da altri linguaggi orientati agli oggetti come lo Smalltalk e l'Objective-C (interfacce e risoluzione dei metodi dinamici).

In Java i principi della programmazione orientata agli oggetti sono stati portati alle conseguenze estreme in quanto il codice è costituito esclusivamente da oggetti di classi che comunicano tra di loro mediante messaggi, ed è stato abbandonato il sistema misto del C++, dove, con i costrutti propri dell'OOP, coesistono ancora altri di tipo procedurale (questo in C++ era una conseguenza della voluta compatibilità retroattiva con il C, alla quale in Java si è infine rinunziato). Non esistono più funzioni indipendenti, ma solo metodi nell'ambito di classi, e le variabili dei tipi di base (`int`, `long`, `float`, etc.), pur ammesse nel linguaggio come entità a livello sub-classe, possono esistere solo come membri di classi.

In quanto linguaggio orientato agli oggetti il Java offre i vantaggi propri di questo tipo di programmazione, quali **l'incapsulamento dei dati**, secondo il quale i dati interni degli oggetti sono nascosti all'esterno, il **polimorfismo**, che consente di utilizzare un nome generale per metodi diversi che facciano cose analoghe anche se con qualche diversità a seconda dei casi, e l'**eredità**, mediante la derivazione di nuove classi da altre già esistenti, che favorisce la ***riutilizzazione del codice***.

Per quanto riguarda la **semplicità** ricordiamo che gli sviluppatori di Java si sono prefissi l'obbiettivo di fare di esso un linguaggio di piccole dimensioni (anche se nelle ultime versioni si è via via ingrandito), semplice da scrivere, compilare, farne il debug, e facile da apprendere.

Il Java è infatti più facile da imparare ed usare rispetto al C++ (e di conseguenza con esso si riscontra una maggiore produttività nello sviluppo) in quanto è stato reso più semplice, essendo stato depurato di vari aspetti delicati che tendono a rendere il C++ complesso e a volte anche poco sicuro, come, da un lato sotto il punto di vista di linguaggio OOP, l'*overloading* degli operatori (che fa perdere leggibilità al codice C++) e l'eredità multipla di una classe da più di una classe (una classe può invece in Java derivare da una sola altra classe, la sua superclasse), ma d'altro lato soprattutto di certi aspetti che il C++ ha ereditato dal C, con il quale lo si era voluto compatibile all'indietro, ma che ne hanno diminuito la natura di liguaggio OOP, come le macro, la conversione implicita di tipo dei dati (in Java è imposta invece l'obbligatorietà del *casting* esplicito tra variabili di tipo diverso usate nella stessa espressione), e soprattutto i puntatori e la loro aritmetica tipica del C, che rappresentano generalmente la maggior causa di errori nel C e nel C++. Gli array in Java sono, contrariamente a quelli del C e del C++, tipi di dati a pieno diritto, contenuti in un intervallo di memoria ben definito e sottoposti a controllo sugli indici che non permette di oltrepassarne i limiti. Java è un linguaggio fortemente tipizzato (è cioè, come si dice, *strongly typed*) e infatti vieta il casting tra oggetti qualsiasi, il casting essendo permesso solo tra variabili elementari numeriche e tra subclassi e superclassi dello stesso oggetto. Queste modifiche non hanno dopo tutto comportato una grande perdita di potenza.

Un altro aspetto importante in Java è quello della gestione della memoria che in esso è resa intrinseca nel linguaggio stesso per cui il programmatore viene sollevato dal problema della liberazione della memoria di volta in volta allocata, grazie al servizio ad alto livello della cosiddetta ***garbage collection***, che è un'operazione che dealloca automaticamente la memoria già riservata per variabili ed oggetti quando essi non sono più adoperati in un programma. L'allocazione di memoria e la garbage collection automatiche semplificano la programmazione e migliorano l'affidabilità rispetto al C e C++, dove i programmatori dovevano scrivere i loro propri gestori di memoria, causa spesso di errori e con conseguenze importanti sulle prestazioni. Bisogna però riconoscere che questa funzionalità, che evita i rischi di *memory leakage* (memoria non recuperata dopo l'uso), va a scapito tuttavia dell'efficienza all'esecuzione perché comporta una maggiore complessità del sistema di run-time, anche se il *garbage collector* di Java lavora come thread in background e quindi agisce quando il tempo del processore diventa disponibile, così da interferire il meno possibile sull'esecuzione del programma.

Il linguaggio Java possiede una libreria di classi per la gestione del protocollo di rete TCP/IP (***T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol*) mediante la quale un programma può accedere alla rete Web per mezzo di protocolli come l'HTTP *(HyperText Tr.Pr.)* o l'FTP *(File Transfer Protocol)*, e quindi utilizzare per esempio dati da un sito Www tramite il suo indirizzo URL (***U**niform **R**esource **L**ocator*). Per questo Java è considerato un **linguaggio per applicazioni distribuite**, cioè di tipo client/server nel senso che permette la condivisione delle informazioni e la collaborazione delle applicazioni in rete. C'è da dire che questo suo aspetto ha reso assolutamente necessaria l'adozione in esso di **misure di sicurezza** in relazione al rischio di operazioni, intenzionali o involontarie, che possano causare danni al sistema client, come la modifica o la cancellazione di file o di altri dati o l'introduzione di virus: ciò è ottenuto con il controllo rigoroso da parte del sistema di run-time degli ***accessi ai servizi del sistema*** (ad esempio operazioni su file e directory) e il ***divieto di accesso diretto alla memoria*** del sistema (allo heap e allo stack). Eventuali tentativi di accesso alla memoria sono individuati e bloccati dal sistema di run-time, che controlla in generale anche altri tipi di infrazione, come l'eventuale mancanza di conversione esplicita (*casting*) tra dati di tipo diverso (il casting implicito non è ammesso dal linguaggio), la presenza di referenze illegali*[Byte#]* o altri errori grammaticali. Pure sul caricamento di classi, effettuato dinamicamente al momento dell'esecuzione (*run-time*), viene eseguito un controllo mediante la classe `ClassLoader` facente parte del sistema. Certi

controlli sono ripetuti anche quando il compilatore li avrebbe dovuti già effettuare al momento della generazione del bytecode, perchè non si esclude la possibilità che il compilatore non fosse affidabile.*[Tutti a Java, BIT, Marzo 96]*

La **portabilità** e l'indipendenza dalla piattaforma (cioè l'aspetto *cross-platform*) di Java, di cui si è parlato nella parte introduttiva di questo capitolo, è perseguita sia a livello del codice sorgente (in questo senso un po' d'altra parte come il C), sia a livello di codice binario compilato (bytecode): ciò vuol dire da un lato che un programma si scrive nello stesso modo qualunque sia il sistema sul quale viene sviluppato, e dall'altro che, una volta compilato su una determinata piattaforma, il medesimo file binario può essere poi eseguito mediante un interprete specifico su qualunque altra piattaforma senza necessità di ricompilazione, con tutto il comprensibile vantaggio che questo viuene ad offrire agli sviluppatori. In Java la portabilità a livello sorgente è assicurata anche dall'adozione in esso degli *standard IEEE* per i tipi di dati di base (o primitivi), secondo cui per esempio un `int` è sempre rappresentato a 32 bit, mentre in C e in C++ la dimensione in memoria di un `int` dipende dalla macchina e dal compilatore. Un altro aspetto regolato in Java è quello relativo all'ordine dei byte. Anche l'ambiente di run-time è portabile in quanto scritto in ANSI C. La portabilità poi dell'interfaccia grafica utente è assicurata dalla standardizzazione POSIX adottata dal linguaggio.

Il problema dell'aggiornamento delle librerie di classi fornite da terze parti, che nel C++ può comportare a volte la necessità di aggiornare anche il codice che ne fa uso, viene affrontato in Java con il cosiddetto ***binding dinamico*** mediante le interfacce con le quali sono specificate le interazioni tra oggetti.

Java ha capacità intrinseche di **multithreading** in quanto nel linguaggio è prevista la possibilità di creare e trattare diversi *thread* (processi leggeri) indipendenti nell'ambito di una programmazione concorrente ottenuta mediante una programmazione semplice ed affidabile (queste caratteristiche sono state ricavate in gran parte dal sistema *Cedar-Mesa* precedentemente sviluppato da Xerox). Questa caratteristica è importante perchè oggi c'è la tendenza ad usare sempre più negli applicativi diversi thread. Un limite a questa potenzialità può derivare soltanto da eventuali limiti insiti nella piattaforma stessa adoperata, la quale potrebbe anche non supportare i thread paralleli.

1.6 Sicurezza delle applet

Le applet, che vengono scaricate (*downloaded*) dalla rete e sono eseguite sul sistema locale del visitatore Web, vengono a sollevare varie preoccupazioni riguardo alla sicurezza, in quanto si può temere che esse possano (involontariamente o deliberatamente) produrre danni sul sistema che le ospita per mezzo del browser. Una prima risposta a questo problema è fornita dai browser stessi che impongono alle applet delle severe restrizioni impedendo che esse possano leggere, scrivere o manipolare file sul disco del sistema locale, che possano eseguirvi dei programmi, accedere ad una stampante o stabilire delle connessioni verso altre macchine diverse dal server stesso da cui sono state emesse.

Un altro accorgimento che riduce i pericoli, senza ancora eliminarli completamente, è quello di effettuare dei controlli sia a livello del compilatore sul codice sorgente, sia a quello dell'interprete sul bytecode al momento dell'esecuzione (non è detto che una volta che il codice sia passato al compilatore non si abbiano poi errori all'esecuzione o run-time).

Soluzioni a questo problema sono comunque continuamente in studio e applicate per ridurre sempre più il pericolo di applet *ostili*.

Le applet possono essere **trusted** e **untrusted**.

1.7 Network (Java) Computing

La Sun prevede come risultato della tecnologia Java l'affermarsi di quello che essa chiama il ***Java Computing*** (o ***Network Computing***) per l'informatica aziendale [1.7]. Con questa soluzione la maggior parte dell'attività si sposterebbe dal desktop (personal computer di tipo tradizionale) alla rete e ai server e da questo conseguirebbe da un lato una semplificazione strutturale per le macchine client e dall'altro una centralizzazione e razionalizzazione dell'attività sui server. Così verrebbe superato il modello aziendale attuale di prevalente attività sui desktop, che comporta per gli utenti complessità, necessità di amministrazione ed aggiornamento continuo, poca sicurezza e affidabilità, oltre ai problemi conseguenti alla molteplicità di piattaforme coesistenti e a tempi di sviluppo di applicazioni lunghi. I benefici di una tale architettura sarebbero un costo annuo totale per postazione (CO#) ridotto, la condivisione di tutte le funzionalità da parte di tutti gli utenti e l'unificazione dell'interfaccia.

Le macchine client potrebbero essere dei client leggeri (*thin client*) basati su Java, facenti parte di una struttura di intranet aziendale. Un'*intranet* non è altro che una rete aziendale funzionante con tecnologie Internet standard (protocollo di trasmissione TCP/IP e altri protocolli, come HTTP, NFS, SNMP, etc.) per operazioni di tipo informativo o transazionale. L'intranet può poi essere collegata all'esterno ad Internet attraverso un *firewall*. I sistemi cosiddetti *legacy*, cioè quelli preesistenti, basati su tecnologie a concezione tradizionale, potrebbero essere usati con server Web funzionanti opportunamente da intermediari (*gateway*) per essi.

I client leggeri sarebbero macchine molto più semplici degli attuali desktop, sui quali il server può scaricare dinamicamente attraverso la rete il codice delle applicazioni, e potrebbe essere il server ad occuparsi dell'archiviazione dei file prodotti o elaborati dal cliente. In una visione totalmente Java il sistema operativo potrebbe poi anche essere l'OS Java che è molto semplice, magari memorizzato in una flash ROM o caricato direttamente dalla rete.

[1.7] Bud Tribble, Java Computing per l'azienda, Sun Microsystems, 1996.

Esempi concreti di client leggeri, detti anche ***network computer*** (NC) sono già presenti sul mercato, come la *JavaStation* di Sun e la *Network Station* di IBM.

Oracle è impegnata nello sviluppo della sua **Network Computer Architecture (NCA)** basata su Oracle 8 e Java, che prevede un server e clienti leggeri. [#]

1.8 Progetto San Francisco

IBM ha lanciato il **Progetto San Francisco**, su cui lavorano oltre duemila programmatori, che utilizza Java per lo sviluppo di applicazioni gestionali di impresa e che dovrebbe unificare via via tutto il software della società oggi differenziato in modo spesso incompatibile su piattaforme diverse.

1.9 Java e gli oggetti distribuiti

Per avere un sistema client/server vero e proprio su rete Internet (o intranet) non bastano le tecniche di interattività in modalità CGI, nonostante gli sforzi che sono stati fatti da più parti per migliorare caratteristiche e prestazioni di questa tecnologia, come le estensioni *ISAPI* (sui server Microsoft), *NSAPI* (sui server Netscape) o la tecnologia *ASP (Active Server Pages)* di

Microsoft, tecnologie tutte ancora fondate sull'uso dei form e del protocollo di trasmissione HTTP, con tutto ciò che questo comporta dal punto di vista della lentezza, dell'ingombro e della mancanza di persistenza dello stato, in quanto le applicazioni CGI ripartono ogni volta ignorando i valori della volta precedente. Una vera svolta in direzione di applicazioni veramente client/server sul Web sarà data invece dalla tecnologia degli **oggetti distribuiti** (**componenti**), che trova una soluzione molto promettente, tra altre possibili, in una tecnica consistente nell'uso di Java per la creazione degli oggetti componenti (JavaBeans) e nella sua associazione a **CORBA** (*Common Object Request Broker Architecture*) come standard per l'interazione tra oggetti. CORBA è una piattaforma di oggetti distribuiti che estende la portata delle applicazioni Java attraverso reti, linguaggi e sistemi operativi e Java è forse il linguaggio più adatto per la creazione di oggetti CORBA. Le applicazioni Java, per la loro natura, si prestano a rappresentare degli ottimi componenti, grazie alla proprietà di adattarsi a tutte le piattaforme e di offrire in più la robustezza che multithreading, garbage collection e gestione degli errori assicurano alle applicazioni create con questo linguaggio, e ad esse CORBA fornisce da parte sua, con il suo protocollo **IIOP** (*Internet Interoperable Orb Protocol*), la trasparenza di rete e il legame con tutti gli altri oggetti distribuiti. La tecnologia degli oggetti distribuiti permette alle applicazioni Internet, intranet o extranet di diventare delle vere e proprie applicazioni client/server a livello business.

L'integrazione di CORBA con Java è oggi cosa fatta in quanto nel JDK 1.2 sono compresi un ORB (*Object Request Broker*) di tipo CORBA (un sottinsieme del Joe di Sun) ed il **JavaIDL** come linguaggio IDL (*Ix Dx Lang*) per la creazione dei legami (stub e skeleton) ORB sugli oggetti. La stessa Remote Method Invocation (RMI) del Java, che poteva già prefigurare un primo standard di interazione tra oggetti in rete (vedi paragrafo 8.2), ma era limitato a Java, e perciò poteva venire a costituire un'anomalia nel campo della standardizzazione degli oggetti distribuiti (interferendo sul terreno proprio dell'OMG, *Object Management Group*, organismo di standardizzazione CORBA), verrà infine fatto confluire in CORBA.

[1.9] Robert Orfali, Dan Harkey, Jery Edwards, CORBA Java and the Object Web, Byte, October 1997, page 95.

1.10 Processori Java (*Java Chips*)

Al problema del miglioramento delle prestazioni a livello dell'esecuzione del bytecode si sta anche cercando di dare una soluzione hardware come alternativa a quella software della compilazione *just-in-time* citata al paragrafo 1.2, della quale questa soluzione richiede tra l'altro meno memoria. La Sun Microelectronics sta infatti sviluppando e mettendo a punto dei nuovi **microprocessori Java** dedicati all'esecuzione diretta del bytecode Java, che permettono di evitare il processo di interpretazione, e sono molto più veloci di una *Java Virtual Machine* di tipo software che funzionasse su microprocessori normali, in quanto il set di istruzioni nativo di questi processori è lo stesso bytecode Java. Una tale tecnologia potrebbe dare grande impulso allo sviluppo di dispositivi Java, dai *PC Web* (PC leggeri orientati ad un uso come client Web) ai *Personal Digital Assistant* (PDA) e ai telefoni cellulari. Il limite di questi processori starebbe nella loro utilizzazione esclusiva con software Java. Questi processori potrebbero però essere usati anche soltanto come coprocessori acceleratori Java.

Tre sono i microprocessori progettati finora da Sun: il **picoJava**, che è un core molto piccolo ed economico da adattare a soluzioni specifiche, il **microJava**, che è un microcontrollore basato sul picoJava, progettato per telecomunicazioni o per altre applicazioni, e l'**ultraJava**, da tre a cinque volte più veloce del microJava, con estensioni multimediali e di grafica 3D, particolarmente adatto per PC Web [1.10].

[1.10] *Byte, April 1996, page 25*

La Sun ha concesso la licenza del picoJava a diversi fabbricanti di processori come Nec, Rockwell Collins Inc., Mitsubishi e Samsung.

Dopo una prima versione del picoJava (che potremo chiamare picoJava I), che poteva processare solo codice Java, è stata sviluppata una seconda versione (il picoJava II), che può processare anche del codice C/C++: infatti, mentre in un primo tempo si era previsto un uso di processori Java con codice esclusivamente Java, si è poi visto che il sistema operativo JavaOS è ancora rimasto in parte non Java, e comunque i sistemi operativi a tempo reale (RTOS, ***Real-Ttime Operating System***) non è previsto per il momento che siano riscritti in Java.

Il primo processore Java di Sun sarà il **microJava 701**. Il vantaggio di questo processore nell'eseguire codice Java rispetto ad un processore non Java sarà di prestazioni da 2 a 3 volte superiori. Le caratteristiche sono: frequenza di clock di 200 MHz, tecnologia CMOS a 0,25 micron, 2,8 milioni di transistor, un bus PCI a 33 MHz ed un bus di memoria a 66 MHz. Però la sua potenza di 4 watt ne esclude ancora l'uso per dispositivi *hand-held*.

Rockwell Collins Inc. ha presentato un suo processore Java, il **Jem1**, non basato sul picoJava.

[] Robert McMillan

1.11 JavaOS

JavaOS è un sistema operativo che consiste in una VM che può funzionare su una macchina senza altro sistema operativo e che può eseguire soltanto applicazioni o applet Java. Esso fornisce le librerie, per esempio quelle grafiche del sistema a finestre o di rete, che le applicazioni Java richiedono.

1.12 Uso di Java per sistemi diversi dai PC

1.12.1 Piattaforma PersonalJava

Sun propone l'uso di Java anche al di fuori del settore dei computer tradizionali di tipo desktop. Javasoft ha già rilasciato le specifiche preliminari (versione 1.0) dell'**API PersonalJava** (qui e nel seguito di questo testo la parola API non indica altro che una libreria di classi), che è un sottoinsieme del JDK 1.1 con alcune aggiunte rivolte ai dispositivi di consumo. La piattaforma PersonalJava di Sun è stata concepita infatti per far girare delle applicazioni Java su piccoli dispositivi elettronici che possono anche essere collegati alla rete, come i computer da tenere in mano (*hand-held device*) detti anche ***personal digital assistant*** (PDA), le ***set-top box*** per TV, le console dei videogiochi, i *Web phone* ed i telefoni intelligenti, i cellulari e i cercapersone (*pagers*).

La macchina virtuale del PersonalJava e le classi di supporto dovrebbero potere essere contenute in una ROM di 2 MB e richiedere per l'uso da 1 a 2 MB di RAM. L'ambiente grafico AWT è esteso con nuove API, per funzionalità come per esempio il *double buffering* delle immagini tracciate in memoria prima di essere inviate sullo schermo [1.11.1.1].

La stessa compagnia Sony, gigante dell'elettronica di consumo, con un accordo di licenza, ha scelto questa piattaforma per l'ambiente di rete di intrattenimento *"home"* con prodotti audiovisivi (AV) digitali, principalmente per le sue caratteristiche interpiattaforma, e collaborerà con la Sun allo sviluppo di applicazioni basate sulla tecnologia Java in questo settore.

Object Design ha inoltre già sviluppato un database, ObjectStore PSE Pro for Java, al 100% puro Java, per la piattaforma PersonalJava [1.11.2].

Sun ha anche creato Personal WebAccess, un browser Web compatto (solo 750 kB) e ricco di funzionalità, costruito come collezione di componenti JavaBean, per i dispositivi che utilizzano la piattaforma PersonalJava, che può essere personalizzato nell'aspetto (*look and feel*) e al quale si possono integrare altre applicazioni [1.11.1.3].

[1.11.1.1] Tom R.Halfhill, Java gets down to business, Byte, October 1997, page 87.

[1.11.1.2] Object Design Announces Industry's First Java Database for PersonalJava Applications, http://www.prnewswire.com/cgi-bin/stories.pl?ACCT=105&STORY=/www/story/09-14-1998/0000750003&EDATE=

[1.11.1.3] Personal WebAccess, The Web Browser for PersonalJava Devices, http://java.sun.com/products/pwa/.

1.12.2 EmbeddedJava

Java si rivela interessante per essere anche applicato sui sistemi "*embedded*", sistemi cioè con processore incorporato, per le stesse ragioni per le quali esso è indicato per le applicazioni desktop o server. L'API Java Application Environment (JAE) **Embedded Java**, predisposta per questi dispositivi, compatibile verso l'alto con PersonalJava e Java, è caratterizzata da un piccolo ingombro di memoria e funzionalità limitate in relazione ai dispositivi sui quali deve essere applicata. I dispositivi cui questo può essere applicato vanno dai telefoni mobili ai pager, dai controllori industriali di processo alla strumentazione e apparecchiature di test e misura, dalle periferiche d'ufficio e stampanti ai dispositivi di rete, come router o switch, dai dispositivi medici alle apparecchiature per infrastrutture di telefonia.

Le applcazioni EmbeddedJava girano su sistemi operativi in tempo reale (Real-Time Operating System o RTOS) e sono ottimizzare per andare su dispositivi aventi piccola capacità di memoria e display di vario tipo. L'API è facilmente portabile ad ogni sistema operativo real-time. Il software è modulare, configurabile e scalabile, e permette di tralasciare caso per caso quelle parti non necessarie in relazione al dispositivo particolare che lo utilizza.

1.12.3 Versioni più leggere di Java

La Sun Microsystems è in contatto con Matsushita per una possibile versione di Java per applicazioni elettroniche di consumo a microprocessore incorporato ancora più leggera (con piccolo ingombro di memoria) di quella *Embedded Java*. Un tale sistema dovrebbe poter occupare lo spazio attualmente occupato nell'industria di elettronica di consumo da sistemi, spesso proprietari, a tempo reale molto leggeri. Le applicazioni che potrebbero beneficiare di questa tecnologia dovrebbero andare dai dispositivi di programmazione televisivi allo *home shopping*, dalle consultazioni mediche al controllo remoto degli elettrodomestici.

1.12.4 JavaCard

Carte a microprocessore o *smart card* (carte intelligenti), più tecnicamente chiamate carte a circuito integrato (*Integrated Circuit Card* o ICC) tipo carte di credito contenenti microprocessore e memoria, rappresentano un'evoluzione dal punto di vista della sicurezza e della capacità rispetto alle carte a banda magnetica del tipo delle carte di credito ordinarie. Una

smart card contiene tipicamente un processore a bassa potenza (per esempio l'Hitachi H8/300) e 16 kByte di memoria (principalmente EEPROM) e può ospitare diverse applicazioni contemporaneamente.

[1.12.4.1] Edward K.Conklin, Smart Card and the Open Terminal Architecture, Dr.Dobb's Journal, December 1998, pages 70-80.

La tecnologia **JavaCard** permette di installare una Java VM su carte a circuito integrato tipo *smart card* per un uso di identificazione personale, sicurezza, assistenza medica, carte di credito/debito, commercio elettronico, GSM. L'input e l'output può essere a contatto o anche senza contatto.

Questa Java Virtual Machine è ottimizzata per un hardware tipico di una smart card che al momento attuale è a 8 bit e con RAM molto ridotta. In essa sono previsti solo interi di tipo `short` o `byte`, gli oggetti sono persistenti tra una sessione e l'altra in una memoria non volatile, non c'è garbage collection, né multithreading, le applicazioni possono essere autenticate.

L'API Java Card è attualmente alla versione 2.0.

A questa tecnologia sono interessati tutti i maggiori fornitori di smart card nei tre settori principali di telecomunicazioni, banche/finanza e tecnologia dell'informazione (IT). Per le telecomunicazioni è stata definita la specifica particolare per un'API GSM. E' stato costituito il consorzio Java Card Forum tra i licenziatari di Java Card.

1.12.5 Altre API Java

Sun sta lavorando ad altre API specifiche come **JavaTV** per la televisione via cavo, **AutoJava** per l'automobile e **JavaPhone** per il telefono.

1.12.6 JavaPC

JavaPC è una Java Virtual Machine per DOS e Windows 3.1, che permette di utilizzare con Java dei personal computer aventi prestazioni ormai superate.

1.12.7 J/Direct

J/Direct di Microsoft permette alle applicazioni Java di accedere direttamente alle API Windows.

1.12.8 Java nella telefonia mobile

La società Symbian, costituita da Psion con le più importanti compagnie di telefonia mobile, come Nokia, Ericsson e Motorola, per lo sviluppo di software per i PDA (*Personal Digital Assistant*), i telefoni intelligenti ed i comunicatori, ha sviluppato il sistema operativo aperto **EPOC** a 32 bit per dispositivi mobili con ROM. Tale sistema è stato scritto in C++, che è quindi il linguaggio nativo delle sue API, e solo minimamente in assembler. Ma, benché l'uso del C++ per lo sviluppo rappresenterebbe la migliore soluzione dal punto di vista delle prestazioni, EPOC fornisce due opzioni per lo sviluppo rapido di applicazioni, una in OPL (*Organiser*

Programming Language, sorta di BASIC, proprio dei dispositivi *personal organiser* Psion) ed una in Java (considerato questo uno standard industriale, orientato agli oggetti, con un vasto supporto di tool di sviluppo), che supporta il JDK 1.1 e prevede un'interfaccia grafica AWT sulla GUI di referenza di EPOC chiamata Eikon.

L'SDK Java per EPOC permette la costruzione con normali ambienti di sviluppo Java di applicazioni Java che si presentano con l'aspetto di applicazioni EPOC native. L'interfaccia grafica Eikon è prevista per dispositivi di base con input a penna o a tastiera e con uno schermo LCD metà VGA (640x240) a scala di grigi, ma può essere adattato per dispositivi con caratteristiche diverse, perchè i dispositivi che ospitano EPOC possono essere i più vari (per esempio l'Ilium Accent di Philips Consumer Communications, PCC, che pure adotta EPOC, ha uno schermo di 640x200 e non ha tastiera, o meglio ha una tastiera su schermo), e perciò le applicazioni EPOC devono distinguere tra il motore e l'interfaccia grafica.
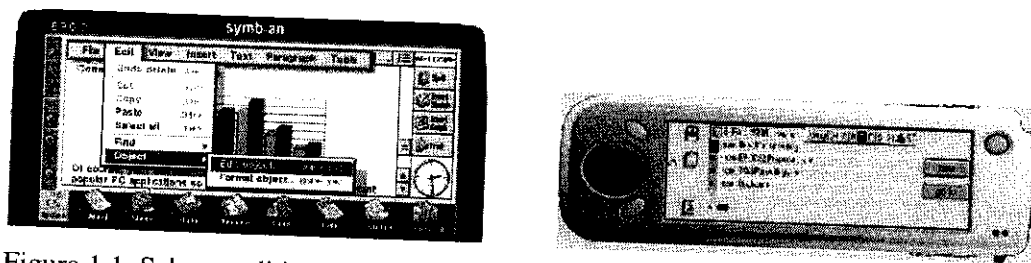
Figura 1.1 Schermo di interfaccia utente di EPOC tipo Eikon e tipo Accent.
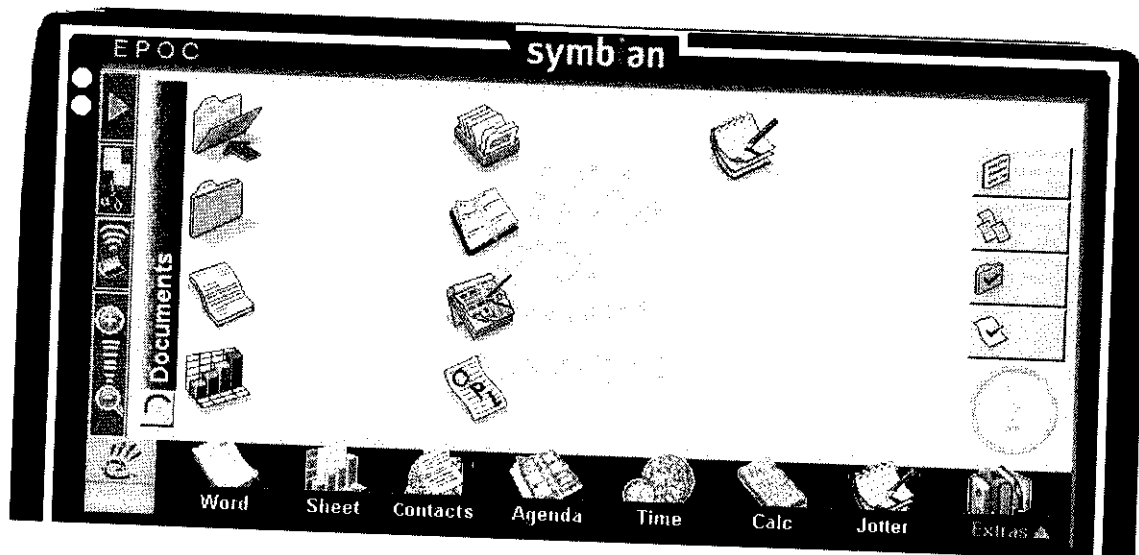
Figura 1.2 Interfaccia EPOC

[1..1.5] Martin Tasker, EPOC Overview, White Papers, EPOC Technology, Symbian, June 1998, http://www.software.psion.com/tech/papers/overview/overview.html.

1.12.9 Tecnologia Jini

Jini è un'architettura di computing distribuito basata su Java e sviluppata da Sun (progetto *JavaSpaces*). Questa tecnologia permette ai più vari dispositivi di collegarsi in rete, registrarsi ed essere accessibili da altri dispositivi collegati alla rete stessa (si parla in tal senso di una "federazione" di dispositivi elettronici). Per l'interazione tra due dispositivi collegati che comunicano tra di loro quello che viene richiesto è la presenza di una Java Virtual Machine *Jini-capable* su ciascuno di essi. Jini è costituito da un leggero strato di software posto sopra le applicazioni Java, che permette il trasferimento sulla rete del codice di applicazioni. Jini comprende un insieme di librerie di classi (Java) e si basa sulla tecnica RMI di Java. Il codice di Jini è stato mantenuto nei limiti ristretti di 48 kB (con un nucleo di 25 kB per i dispositivi più semplici) in modo da poter essere introdotto in ogni dispositivo suscettibile di essere collegato in rete. Jini dovrebbe facilitare la connettività (dovrebbe bastare inserirne la spina del dispositivo alla rete) e la condivisione in rete di periferiche *Jini-enabled* (per esempio una stampante) indipendentemente dal sistema operativo e dal protocollo. I dispositivi possono andare dai dischi rigidi per l'immagazzinamento di dati alle videocamere digitali, dai PC o processori ai display, dalle stampanti e scanner ai lettori CD e DVD, dai telefoni ai videoregistratori e TV, e perfino anche agli elettrodomestici. Jini fornisce un servizio di *look-up* che raccoglie le informazioni sulle caratteristiche che ogni dispositivo dichiara di avere al momento in cui viene collegato alla rete e si mette a disposizione di tutti gli utilizzatori connessi autorizzati. Per esempio un disco rigido può con Jini rendersi direttamente disponibile su una rete al di fuori di un PC e di un sistema operativo (ecco così spiegato il particolare interesse mostrato per Jini dai fabbricanti di memorie di massa). Diverse compagnie, come Sony, Toshiba, Canon, Mitsubishi, Epson e Oki (stampanti), Quantum e Seagate (dischi rigidi), Ericsson (telefoni cellulari), Novell (reti), Computer Associates, si sono mostrate interessate a questa tecnologia. Sun sta studiando (Mitra) la possibilità di usare con Jini delle periferiche che non dispongono di una loro JVM per mezzo di dispositivi specifici che collegati alla periferica agiscano come JVM fornendo ad essa questa funzionalità.

[1.11] Jason Krause, Rob Guth, Sun turns Java into Jini, IDG-net, July 15, 1998.

[1.11] Jon Byous, Jini Technology Grants The Ultimate Wish, The Source for Java Technology, www.javasoft.com, July 20, 1998.

1.13Bibliografia

[1] Ed Tittel, Mark Gaither, Java, Apogeo, 1996. Traduzione in italiano del testo: "60 Minute Guide to Java" degli stessi autori, IDC Books Worldwide Inc., 1995.

[2] Elliotte Rusty Harold, Brewing Java: A Tutorial, http://sunsite.unc.edu/javafaq/javatutorial.html, 1996.

[3] John December, Java Immagini in movimento nelle pagine WWW, Tecniche Nuove, 1996. Traduzione in italiano del testo "Presenting Java - An Introduction to Java and HotJava", Sams.net Publishing, Indianapolis, IN (USA), 1995.

[4] Tim Ritchey, Usare Java, Jackson Libri, 1996. Traduzione in italiano del testo "Java!", New Riders Publishing, 1995.

[4] Laura Lemay, Charles L. Perkins, Teach Yourself Java in 21 Days, Sams.net Publishing, Idianapolis 1996.

[5] Mark C. Reynolds, Java Programming From the Grounds Up, Mecklermediàs iWorld, http://pubs.iworld.com, Spring 96 issue.

[6] Andrew Singleton, A Hot Cup of Java, Byte, April 1996, pages 129-132.

[7] Tom R. Halfhill, Java Chips Boost Applet Speed, Byte, April 1996, pages 129-132.

[8] Tom R. HalfHill, Arriva il Web-PC, Bit, Aprile 1996, pag. 50-63.

[9] Michele Constabile, Tutti a Java, Bit, Marzo 1996, pag. 64-66.

[10] Al Stevens, Teach Yourself... C++, MIS:Press, 1990.

[11] Donald G. Drake, Introduction to Java threads, JavaWorld (magazine), April 1996.

[12] Laura Lemay, Charles L. Perkins, Teach Yourself Java in 21 Days, Sams.net Publishing, Indianapolis, 1996.

[13] Art Taylor, JDBC Developer's Resource, Informix Press.

[14] Ashton Hobbs, Teach Yourself Database Programming with JDBC, Sams.net, 1997.

[15] Cliff Berg, How do I Write a Java Servlet?, Dr.Dobb's Journal #270, October 1997, page 121.

[16] Al Stevens, JavaScript and CGI, Dr.Dobb's Journal, April 1997, pag. 92-97.

[17] Frnk Buss, Stefan Schlöpke, Programmation Java en 15 heures chrono, Micro Application, Paris, 1997.

[18] Michel Martin, Java 1.1 Guide du développeur, Sybex, Paris, 1997.

[19] Cédric Nicolas, Christophe Avare, Frédéric Najman, Java client-serveur, Editions Eyrolles, Paris, 1997.

[20] The GNU C Library Sockets.

2. Java come linguaggio orientato agli oggetti

2.1 Classi

2.1.1 Classe e sua definizione

Si è già detto che Java è un linguaggio orientato agli oggetti (*Object-Oriented Programming o OOP*), è un linguaggio cioè che, invece che su variabili e funzioni indipendenti singolarmente considerate in un programma, si basa su una simulazione del mondo reale fatta attraverso elementi più complessi rappresentati dagli oggetti con le loro interazioni mutue. Gli **oggetti** sono entità introdotte in un programma costituite da un insieme di ***dati*** uniti insieme, dati che possono essere paragonati a quelli di una struttura del C o di altri linguaggi, e che in generale rappresentano le caratteristiche strutturali di un oggetto concreto o astratto del mondo reale che il programma vuole simulare, e da ***funzioni*** (dette metodi) associate e strettamente connesse a tali dati, agenti su di essi (nei linguaggi non-OOP dati e funzioni sono invece indipendenti tra di loro). Oggetti più complessi possono poi essere composti da altri oggetti semplici e così via.

Gli oggetti sono definiti mediante le *classi* cui essi appartengono: una **classe** descrive nella sua struttura e funzionalità il modello di una categoria di oggetti, cioè definisce le caratteristiche generali degli oggetti che fanno parte della categoria e in particolare quali e di che tipo sono i due insiemi di caratteristiche seguenti:

- gli ***elementi costitutivi*** o **attributi**, generalmente detti **variabili d'istanza** perchè sono caratteristiche di ciascuna istanza della classe, o anche a volte **dati membri** della classe o anche a volte campi (*field*), che possono essere dei tipi base (`int, long, float,` etc.) o anche altri oggetti, che caratterizzano le proprietà strutturali (aspetto o stato) dell'oggetto e che, essendo delle variabili, possono assumere valori diversi nel corso del programma in cui l'oggetto può subire delle modificazioni,

- ed i ***comportamenti*** che sono rappresentati da funzioni (in C++ erano chiamate **funzioni membro** della classe, ma secondo la terminologia dell'OOP sono più propriamente dette **metodi**) che manipolano le variabili degli oggetti della classe o che possono compiere delle azioni (interazioni) anche su altri oggetti esterni che siano accessibili.

Attributi e metodi possono essere chiamati con il termine generale di **membri** della classe che li comprende tutti e due.

In parole semplici ed intuitive si può dire che i dati membri di una classe descrivono attraverso il loro valore *come è fatto* concretamente ed *in che stato si trova* un particolare oggetto e i metodi invece *come esso si comporta* o *le operazioni che può fare*. Nella classe si definisce in modo astratto *quali* e *di che tipo* sono le variabili d'istanza di ogni oggetto che faccia parte di quella classe, e poi ogni oggetto concretamente creato dalla classe stessa presenta i suoi propri reali *valori* di queste variabili, che non è detto siano costanti ma possono variare nel corso del programma.

Una classe rappresenta un modello (schema) concettuale e astratto (o una categoria) di un certo tipo di oggetti. Con la **definizione** di una classe si definisce soltanto come è fatto e come si comporta una certa categoria di oggetti, ma non se ne crea ancora alcuno e non se ne alloca la memoria: sarà visto al paragrafo seguente come creare (o, come si dice, ***istanziare***) un **oggetto** reale come un'**istanza della classe** che assume le caratteristiche proprie della classe e dispone la

relativa allocazione di memoria (per esempio `Mario` può essere un oggetto, o istanza, della classe `uomo`: si potrebbe vedere l'oggetto come un'incarnazione della classe).

Un programma ad oggetti può essere visto come un'associazione di diversi oggetti, ciascuno dei quali risponde ad uno schema di principio definito in una classe, con le caratteristiche e i comportamenti in essa definiti, oggetti che possono interagire attraverso i metodi, ma che già al momento in cui sono costruiti fanno quello che è definito in un particolare metodo che è il loro metodo costruttore.

Un programma Java consiste nella definizione di una o più classi.

Una classe viene creata mediante la parola chiave **`class`** posta davanti al nome scelto come identificatore della classe, con una definizione del tipo:

```
class unaClasse {
    // corpo della classe in cui si
    // definiscono le variabili d'istanza
    // e i metodi
}
```

racchiudendo tutto il blocco di definizione della classe, costituito dalla definizione delle sue variabili e dei suoi metodi, tra due parentesi `{` e `}` , e già un'espressione minima come la seguente:

```
class unaClasse { }
```

rappresenta una classe ben definita e costituisce quindi anche un programma Java pienamente valido, della massima semplicità concepibile, che definisce una classe, anche se essa non ha proprietà e non fa niente, non avendo né variabili nè metodi (anche se, per essere più precisi, essendo, come vedremo meglio in seguito, ogni classe in Java derivata dalla classe di sistema di Java `Object`, possiede già quel minimo di funzionalità che la classe `Object` le fornisce).

In Java non esiste un tipo *struttura*, pur presente nella maggioranza dei linguaggi di programmazione, in quanto tutto ciò che può essere fatto con una struttura può anche essere altrettanto bene espresso con una classe, che è l'unica entità ammessa nel linguaggio.

Nel linguaggio Java di base (*core*) ci sono un certo numero di classi predefinite, con funzionalità generali e che costituiscono un punto di partenza per la definizione da parte del programmatore di nuove altre classi. Se per un programma sono sufficienti le classi del sistema non c'è bisogno di crearne di nuove (salvo quella che costituisce l'applicazione stessa), ma il programmatore generalmente ne definisce anche delle altre per le sue esigenze particolari.

2.1.2 Esempi di classi

Per cominciare a fissare le idee facciamo un esempio di definizione di una classe. Supponiamo di avere a che fare in un programma con una o più scritte (per esempio dei titoli) che noi vogliamo rappresentare come oggetti nel nostro programma: il primo compito che nella programmazione ad oggetti si fissa il programmatore è l'individuazione dei tipi di oggetti, e quindi le classi, che possono rappresentare nel migliore dei modi le cose che il suo programma vuole simulare con le loro caratteristiche e funzionalità. Nel nostro caso si può pensare allora di definire una classe `Scritta` come rappresentante la categoria degli oggetti concreti (le scritte) che interverranno nel nostro programma, avente come caratteristiche strutturali o attributi un testo (l'insieme dei suoi caratteri), la sua grandezza, il suo colore, la sua posizione in un piano (x, y) (tutti non necessariamente costanti, ma modificabili durante il programma), e che ha come funzionalità quelle di poter essere creata, scritta, cancellata o posizionata, per mezzo di metodi particolari specifici di ognuna di queste operazioni.

Cominciamo quindi a individuare in modo dettagliato le variabili d'istanza della classe come attributi che possono caratterizzare la nostra classe, che possono essere:

- il *testo* della scritta come stringa (un oggetto della classe `String` della libreria di Java),
- l'*altezza* della scritta (data da un valore intero),
- le coordinate *x* e *y* della posizione della scritta (date come una coppia di valori interi),

e poi i metodi della classe, che permettono di effettuare le seguenti operazioni:

- *creare e inizializzare* la scritta (un metodo per costruire ogni oggetto specifico),
- *scrivere* la scritta,
- *cancellare* la scritta, cioè farla sparire,
- *posizionare* la scritta nel piano.

A questo punto possiamo già accingerci a rappresentare tutto questo e scrivere in un file di testo il nostro codice Java, che consiste nella definizione della classe[2]:

```
// B01Scritta.java (F.Spagna) Primo esempio di una classe: la classe Scritta

public class B01Scritta {
   String testo;         // variabile che rappresenta il testo della scritta
   int altezza;          //     "       "       "      l'altezza della scritta
   int x;                          // variabile coordinata x della posizione
   int y;                          // variabile coordinata y della posizione

   public Scritta() {      // metodo che costruisce e inizializza un oggetto
      x = 100;             // con coordinate di default prefissate (100,100)
      y = 100;
   }                               // secondo metodo costruttore con argomenti
   public Scritta(String tex, int alt, int X, int Y) {
      testo = tex;                      // valore delle variabili di istanza
      altezza = alt;               // introdotte come argomenti del costruttore
      x = X;
      y = Y;
   }
   public scrive(int X, int Y) { // scrive la scritta in una posizione data
      // istruzioni del metodo che scrive la scritta
   }
   public void cancella() {                            // cancella la scritta
      // istruzioni del metodo che cancella la scritta
   }
   public void posiziona(int X, int Y) {  // posiziona la scritta nel piano
```

[2] Diciamo subito che gli esempi fatti in questi appunti sono tutti provati e perfettamente funzionanti e il loro codice è presentato in un riquadro nella versione esatta sempre realmente compilata ed eseguita, e, dove è possibile, ne vengono presentati i risultati, sotto forma di file di testo per le applicazioni a carattere, e con cattura dello schermo e memorizzazione del contenuto del clipboard (ambiente Windows 98) in file di formato grafico GIF o JPG) per i programmi in grafica. Si è sempre cercato di ridurre il codice di tali esempi alle dimensioni minime ed essenziali, anche se ciò li rende spesso di scarso interesse pratico, per concentrare tutta l'attenzione sull'aspetto di volta in volta esemplificato, limitando tutto quello che altrimenti potrebbe distrarre il lettore. In qualche caso tuttavia è stato ritenuto interessante fare degli esempi un po' più allargati che presentano un aspetto un po' più realistico. I sorgenti sono tutti disponibili presso il sito Internet http://apple.arcoveggio.enea.it:8069/java.htm con gli stessi nomi delle classi presentate nel testo. La numerazione posta in testa ai nomi di classi e file, costituita da una lettera che rappresenta il capitolo e da un numero progressivo relativo al capitolo stesso, pur appesantendo un po' il testo, dovrebbe essere utile per la manipolazione e la prova degli esempi.

```
         x = X;
         y = Y;
      }
   }
```

Per quanto riguarda le variabili d'istanza di quest'esempio per il momento non notiamo niente di particolare, mentre per i metodi, sui quali ci soffermeremo in seguito, osserviamo qui solo che essi sono strutturati con un'intestazione che comprende eventuali modificatori (nel caso nostro la parola `public`), tipo del valore di ritorno, nome del metodo, parentesi `()` contenenti eventualmente una lista di parametri ciascuno col proprio tipo, seguita dal corpo del metodo contenente il blocco delle istruzioni interne al metodo racchiuse tra due parentesi `{` e `}`.

Per quanto riguarda i modificatori notiamo qui appena che il termine `public` scritto davanti al nome dei metodi della classe serve per rendere quei metodi accessibili (o, se si vuole usare un'altra espressione, *visibili*) dai metodi delle altre classi, ma su quest'argomento torneremo al paragrafo 2.1.7.

Il tipo di un metodo è il tipo del valore che il metodo restituisce cioè di quel valore che, definito in un'istruzione `return` del metodo, viene messo nel programma nel punto di invocazione del metodo quando questo è eseguito (se non restituisce nulla si scrive `void`).

Il primo metodo, che porta lo stesso nome della classe, è detto **costruttore** (*constructor*) in quanto Java lo manda in esecuzione automaticamente al momento della creazione di ogni oggetto della classe (usando l'operatore `new`, come vedremo meglio nel paragrafo seguente).

La classe, così definita in un file che avrà il nome `B01Scritta.java`, cioè come quello della classe stessa che esso contiene, ma con l'estensione `.java`, può essere compilato con il compilatore Java, mandato in esecuzione dalla linea di comando richiamando il suo nome `javac`:

```
javac B01Scritta.java
```

che produrrà un file `B01Scritta.class` che nel semplice caso dell'esempio non rappresenta ancora né un'applicazione né un'applet, non avendo le proprietà né dell'una né dell'altra, ma può essere utilizzato da un'applicazione o da un'applet che istanzi un oggetto di quel tipo come sua variabile d'istanza o come variabile locale ad un metodo, oppure per derivarne altre classi (e questo è proprio quello che faremo nell'esempio del paragrafo 2.1.11). Se invece si volesse utilizzare una classe come applicazione o come applet bisognerebbe darle certe caratteristiche particolari, che vedremo in seguito (dotarla di un metodo `main()`, come vedremo al paragrafo 2.1.3, per un applicazione, e derivarla da una classe `Applet` per un'applet).

A proposito della scrittura del codice notiamo che l'indentazione, cioè il rientro delle righe di programma via via che si procede nell'annidamento delle istruzioni, pur non essendo obbligatorio, è di uso generalizzato perché favorisce la leggibilità del codice evidenziandone la struttura. Noi siamo propensi ad un'indentazione di multipli di tre spazi, anche se valori di due o di quattro spazi sono anche spesso usati (altri valori molto meno).

Un altro esempio potrebbe essere il seguente, ricavato dal codice sorgente stesso di sistema di Java: si tratta di una versione da noi semplificata (sono stati eliminati alcuni metodi) della classe `Point` (del package `java.awt` di cui si parlerà a suo tempo) di Java, che rappresenta un punto nel senso geometrico:

```
// B02Punto.java  F.Spagna  Un secondo esempio di classe: la classe Punto

public class B02Punto {
   public int x;                                              // coordinata x
   public int y;                                              // coordinata y

   public Punto() {             // costruisce e inizializza un oggetto Punto
      x = 0;                         // con coordinate di default prefissate
      y = 0;
   }
   public Punto(int X, int Y) {         //costr.e inizializ.un oggetto Punto
      x = X;                          // dalle coordinate specificate X e Y
      y = Y;
   }
   public void muove(int X, int Y) {      // riposiziona il punto nel piano
      x = X;
      y = Y;
   }
   public void trasla(int dX, int dY) {        // sposta il punto sul piano
      x = x + dX;
      y = y + dY;
   }
}
```

Senza voler ancora analizzare completamente la definizione, osserviamo intanto che la classe `Punto` da noi dichiarata ha due variabili d'istanza, le coordinate `x` e `y`, e quattro metodi, tra i quali due costruttori `Punto(...)`, e due altri `muove(...)` e `trasla(...)`.

2.1.3 Dichiarazione di una variabile oggetto e creazione di un oggetto

La **creazione di un oggetto** concreto appartenente ad una classe, che viene a costituire un'istanza (un esemplare) della classe (quando si crea un oggetto si dice anche che si *istanzia un oggetto della classe*) con il proprio spazio di memoria necessario allocato, viene fatta mediante l'operatore **new** posto davanti al nome della classe, precisando tra parentesi dopo il nome della classe, nel loro ordine, il valore particolare che si vuole dare agli argomenti richiesti dal costruttore, poiché il new, nel creare l'oggetto, ne invoca automaticamente il costruttore. L'espressione di creazione di un oggetto ha così la forma:

```
new nomeClasse();
```

oppure:

```
new nomeClasse(valoreParam1, valoreParam2, ...);
```

e queste espressioni, oltre a creare l'oggetto, restituiscono una referenza (o, se vogliamo, un indirizzo, un puntatore) all'oggetto creato, con la quale l'oggetto può essere reperito (secondo lo schema fatto qui di seguito).

referenza all'oggetto ←---- `new nomeClasse() .`

Per esempio le espressioni:

```
new Scritta();

new Punto();
```

che si riferiscono alle classi definite negli esempi fatti al paragrafo precedente, creano rispettivamente un oggetto di tipo `Scritta` ed un oggetto di tipo `Punto` usando in particolare nei due casi costruttori senza argomenti.

Un oggetto appartenente ad una determinata classe così creato può essere ***referenziato*** da una **variabile oggetto** (*object variable*) che sia stata dichiarata del tipo della classe cui appartiene l'oggetto e che punta ad esso. La dichiarazione di una variabile referenza di un oggetto viene fatta facendo precedere il nome della variabile dal nome della classe, esattamente come si fa per esempio per una variabile di tipo elementare, ad esempio di tipo intero, che viene dichiarata (vedi paragrafo 3.7) con:

```
int i;
```

Pertanto nel caso delle classi dell'esempio precedente si possono dichiarare una variabile oggetto `scr` della classe `B01Scritta` oppure una variabile oggetto `p` della classe `Punto`:

```
B01Scritta scr;

Punto p;
```

che possono referenziare rispettivamente oggetti di tipo `Scritta` ed oggetti di tipo `Punto`.

Però osserviamo che questa dichiarazione, diversamente da quello che succede per i tipi elementari, per cui la semplice dichiarazione riserva già la memoria relativa, per un oggetto dice ancora soltanto di che tipo è l'oggetto referenziabile da `scr` o da `p`, ma non ne crea ancora alcuno, perché è con l'operatore `new` che gli oggetti vengono creati.

L'assegnazione ad una variabile oggetto della referenza ad un oggetto creato con il `new` viene fatta con istruzioni del tipo:

```
String str;         // dichiarazione di una variabile oggetto di tipo String
str = new String(); // creazione di un un oggetto ed assegnaz.alla variabile
                    // della referenza ad ess

Punto p;
p = new Punto();
```

Si può però anche scrivere più sinteticamente, al posto delle due istruzioni separate (rispettivamente dichiarazione di variabile e allocazione dell'oggetto con assegnazione della sua referenza), una sola istruzione comprendente allo stesso tempo le operazioni di dichiarazione di variabile oggetto, creazione dell'oggetto, ed assegnazione della sua referenza alla variabile:

```
String str = new String();

Punto p = new Punto();
```

Tornando agli esempi del paragrafo precedente, ma questa volta usando costruttori che prevedono degli argomenti, si potrebbe scrivere:

```
B01Scritta scr = new B01Scritta("Ciao", 2, 5, 10);

Punto p = new Punto(1, 2);
```

se `"Ciao"` è il testo iniziale della scritta e `2, 5` e `10` i valori iniziali degli argomenti `altezza, x` ed `y` per il primo esempio, e se `1` e `2` sono i valori iniziali di `x` ed `y` per

il secondo esempio. Quando il `new` richiama il costruttore questa volta ne richiama la versione che prevede degli argomenti e ad esso sono passati i valori assegnati agli argomenti (vedi paragrafo 2.1.6.3). In generale in un metodo costruttore di una classe si fanno delle operazioni di inizializzazione, generalmente di variabili d'istanza dell'oggetto, ma può anche essere fatto altro, a discrezione del programmatore, in relazione a quello che egli vuole sia fatto subito all'atto della creazione dell'oggetto.

L'operatore `new` svolge i seguenti compiti:

- alloca (nello *heap*) la quantità di memoria necessaria per l'oggetto che viene creato e ne restituisce una referenza,

- inizializza automaticamente a valori nulli (vedi meglio al paragrafo 2.1.5) ogni variabile d'istanza dell'oggetto,

- richiama automaticamente il metodo costruttore dell'oggetto, che svolge i compiti assegnatigli nel programma a livello della definizione della classe, che sono in generale principalmente operazioni di inizializzazione di variabili d'istanza ed eventuale creazione di oggetti membri, ma che possono anche prevedere la chiamata di altri metodi dello stesso o di altri oggetti.

Un'espressione di istanziazione di un oggetto fatta con il `new` può anche essere passata direttamente in quanto tale come argomento ad un metodo nel punto in cui in esso è previsto un argomento del suo stesso tipo: l'oggetto viene così creato all'atto della chiamata del metodo e la sua referenza passata al metodo attraverso la variabile che rappresenta l'argomento all'interno del metodo, anche se questa referenza non è assegnata ad alcuna variabile fuori del metodo e quindi non è utilizzabile all'esterno di esso:

```
oggetto.metodo(new classeArgomento());
```

ed in tal caso non viene creata una variabile di referenza all'oggetto, ma, se si vuole conservare in una variabile, chiamata ad esempio `ogg`, il valore dell'argomento per utilizzarlo poi anche fuori del metodo, si può fare l'assegnazione della referenza all'atto del passaggio dell'argomento:

```
classeArg ogg;
oggetto.metodo(ogg = new classeArg());
```

e questo è possibile in quanto in Java l'assegnazione restituisce il valore dell'oggetto assegnato.

E' da chiarire bene il fatto che quando si assegna la referenza di un oggetto creato ad una variabile oggetto che è usata per identificare l'oggetto e che è come se lo rappresentasse, in realtà la variabile non memorizza in sé l'oggetto, come succede per le variabili elementari che memorizzano direttamente i valori, ma solo una **referenza** all'oggetto, referenza che è in sostanza un puntatore (o, se si vuole, un indirizzo) all'oggetto, simile per molti aspetti ad un puntatore del C, ma privato della possibilità di una sua gestione diretta. Una variabile inizializzata con la referenza ad un determinato oggetto non è affatto detto che resti sempre collegata con quell'oggetto, ma può nel corso di un programma cambiare di valore per puntare di volta in volta ad oggetti diversi, così come ad una variabile numerica possono essere assegnati nel corso di un programma valori diversi.

Per esempio si può avere:

```
Punto p = new Punto(1, 1);        // p punta ad un certo oggetto
p = new Punto(2, 2);              // p adesso punta ad un oggetto diverso
```

Se poi si dichiara un'altra variabile `p1` ancora di tipo `Punto` e le si assegna il valore di `p`:

```
Punto p1 = p;
```

questo non significa che si crea, e cioè viene memorizzato, un nuovo oggetto `p1` copia dell'oggetto `p`, bensì che `p1` è una seconda variabile che punta allo stesso oggetto referenziato da `p`, cioè rappresenta un'altra referenza allo stesso oggetto. Per avere un nuovo oggetto copia dell'oggetto dato bisogna istanziarlo con un altro `new` (per la copia di oggetti si può anche vedere il paragrafo 2.7).

Per far risaltare meglio la differenza tra l'assegnazione di un valore ad una variabile di tipo elementare e l'assegnazione di una referenza ad una variabile oggetto facciamo un esempio nel quale si vede come, una volta assegnata una referenza `p` ad un'altra `p1`, `p` e `p1` restano agganciati per puntare sempre allo stesso oggetto finchè non vengono riassegnate. Poi è fatto il caso analogo per variabili di tipo elementare, la cui assegnazione reciproca è solo temporanea. Ecco il codice:

```
// B03Refer.java (F.Spagna)  Referenze ad oggetti

import java.awt.Point;

public class B03Refer {

   public static void main(String s[]) {

      Point p = new Point(0, 0);
      Point p1 = new Point(1, 1);
      System.out.println("p=("+p.x+","+ p.y+") p1=("+p1.x+","+p1.y+")"
            + " le 2 referenze, indipendenti, puntano a 2 punti distinti");
      p1 = p;
      System.out.println("p=("+p.x+","+ p.y+") p1=("+p1.x+","+p1.y+")"
            + " p1 posto = p: adesso puntano allo stesso punto");
      p.x = 3;
      System.out.println("p=("+p.x+","+ p.y+") p1=("+p1.x+","+p1.y+")"
            + " p.x e' cambiato ma p1 punta sempre allo stesso oggetto di p");
      p1.x = 4;
      System.out.println("p=("+p.x+","+ p.y+") p1=("+p1.x+","+p1.y+")"
    + " ora p1.x e' cambiato: si vede come anche p1 modifica l'oggetto di p");

      int a = 0;
      int b = 1;
      System.out.println("a="+a+" b="+b);
      a = b;
      System.out.println("a="+a+" b="+b);
      a = 3;
      System.out.println("a="+a+" b="+b);
   }
}
```

Ed ecco i risultati:

```
p=(0,0) p1=(1,1) le 2 referenze, indipendenti, puntano a 2 punti distinti
p=(0,0) p1=(0,0) p1 posto = p: adesso puntano allo stesso punto
p=(3,0) p1=(3,0) p.x e' cambiato ma p1 punta sempre come p
a=0 b=1
```

```
a=1 b=1
a=3 b=1
```

Quindi, non si può memorizzare lo stato di un oggetto ad un certo momento, come si farebbe per una variabile di tipo elementare con:

int x, xold; x = 1; xold=x; x=2; // etc.

perché, se si facesse:

Point p, pold; p = ...; pold=p; // etc.

Pold seguirà sempre l'oggetto p: in tal caso si dovrà invece copiare l'oggetto nell'altro:

Point p, pold; p = ...; p.copy(pold); // etc.[RIVEDI bene#]

Quando una variabile oggetto referenza di un oggetto viene passata come argomento ad un metodo è l'oggetto stesso che viene passato con la referenza e non una copia di esso, e questo è importante perché comporta la possibilità che il metodo modifichi al suo interno l'oggetto (passaggio del parametro per referenza).

2.1.4 Distruzione degli oggetti e *garbage collection*

Come avviene per ogni altra variabile, la referenza ad un oggetto, e con essa l'oggetto stesso, una volta dichiarata, diventa accessibile, o, come si dice anche, ha ***scope***, cioè visibilità, entro il blocco nel quale è stata dichiarata (un **blocco** è ogni porzione di codice compreso tra due parentesi `{` e `}` , un blocco è anche, ed è il caso più frequente, quello che racchiude il corpo del metodo stesso entro cui si presenta la dichiarazione dell'oggetto) e negli eventuali blocchi in questo annidati a livello più interno. Quando il flusso di svolgimento del programma esce dal blocco contenente la dichiarazione, l'oggetto non è più visibile e viene deallocato con un'operazione automatica detta di ***garbage collection*** (che inglese vuol dire "raccolta dell'immondizia"). In Java infatti la gestione della memoria (*memory management*) è automatica ed un *garbage collector* si incarica di ripulire (disallocare) la memoria allocata per un oggetto alla sua creazione, quando riscontra che questo non è più utilizzato (precisamente quando non ci sono più referenze ad esso), cioè quando si esce da ogni blocco in cui esistano in vita referenze all'oggetto.

Ammettiamo di istanziare ripetutamente nel corso di un programma un oggetto sempre con la stessa istruzione (per esempio in un loop):

```
for (int n = 0; n <= 1000; n++)
   p = new Punto(x, y);
```

Ci si potrebbe allora chiedere se bisogna preoccuparsi della memoria occupata da tutti quegli oggetti creati: la risposta è no perchè ogni volta che in un'iterazione del ciclo si crea un nuovo oggetto quello precedente viene dereferenziato (perché la sua referenza `p` ha cambiato oggetto e punta ad uno nuovo) e quindi viene assoggettato alla *garbage collection*.

La *garbage collection* è la ragione per cui in Java non esiste un operatore `delete` per disallocare la memoria degli oggetti come quello del C++, che sarebbe superfluo.

2.1.5 Metodo finalizzatore

Il metodo **finalize()** della classe `Object` di ogni oggetto è chiamato automaticamente dal *garbage collector* prima che l'oggetto venga eliminato dalla memoria. Se si vuole imporre agli oggetti di una classe di compiere qualche operazione al momento della loro distruzione, per liberare risorse di sistema o per svolgere altre operazioni di pulizia, si può quindi ridefinire il metodo:

```
void finalize() { }
```

ed inserire nel suo corpo il codice che si vuole sia eseguito al momento della raccolta dell'oggetto (in generale operazioni varie di ripulitura, come ad esempio togliere le referenze ad altri oggetti non più necessari).

E' da tener presente che il metodo va in esecuzione al momento in cui viene effettivamente deallocata la memoria e non quando ogni referenza all'oggetto è sparita (tra i due momenti può intercorrere un certo tempo, dovuto al modo di lavorare del *garbage collector*) e che il suo funzionamento non è garantito. Normalmente l'uso di `finalize()` non è però necessario.

2.1.6 Variabili d'istanza di un oggetto e variabili di classe

In Java, come deve essere per un linguaggio completamente orientato agli oggetti, esistono solo classi come entità di programmazione e non esistono variabili indipendenti nel programma come le variabili globali di C e C++. Variabili a livello sub-classe (`int`, `long`, `float`, `boolean`, etc.) possono esistere solo come membri di classi (variabili d'istanza), o come variabili dichiarate all'interno dei metodi della classe o come loro argomenti (variabili locali). Esse costituiscono gli elementi più piccoli delle classi, i mattoni elementari.

Le variabili membri di una classe, che possono essere delle variabili elementari di uno dei tipi detti sopra o anche oggetti di altre classi già definite, vengono dette **variabili d'istanza** perché per ogni istanza della classe (cioè per ogni oggetto creato) assumono valori propri. Esse hanno una visibilità generale totale all'interno della classe entro cui si comportano come variabili ***globali*** nel senso che possono essere viste e modificate da qualunque metodo della classe (globalità, notiamo bene, però ristretta all'interno della classe).

Le variabili d'istanza di un oggetto possono essere richiamate con il loro semplice nome dai metodi della classe stessa (e da quelli di ogni altra classe dello stesso package#), per i quali esse sono sempre visibili, in quanto all'interno di una classe, come si è detto sopra, c'è visibilità totale tra i vari elementi. Ma per essere richiamate dai metodi dell'oggetto di un'altra classe devono essere accessibili e perciò essere state dichiarate `public` e si deve far precedere il loro nome da quello dell'oggetto legato con l'operatore punto (`.`) compreso tra il nome dell'oggetto a sinistra ed il nome della variabile a destra. Ad esempio le variabili d'istanza `x` e `y` di un particolare oggetto `scr` della classe `B01Scritta` definita come esempio nel paragrafo 2.1.2 possono essere chiamate da altri oggetti mediante le espressioni `scr.x` e `scr.y`.

Se un oggetto `ogg` ha come variabile d'istanza un altro oggetto `oggMemb`, che a sua volta ha una variabile d'istanza `var`, l'espressione:

```
ogg.oggMemb.var
```

si riferisce alla variabile d'istanza `var` dell'oggetto `oggMemb` che a sua volta è membro di `ogg`. Una catena di nomi di membri di questo tipo può essere lunga quanto si vuole, ma è bene non abusare nella lunghezza di questo tipo di notazione perché via via che cresce essa diventa

meno leggibile. Per facilitare la lettura di tali espressioni teniamo presente che alla sinistra sta il nome dell'oggetto (o della classe se si tratta di elementi statici di classe) contenitore più esterno e via via che si procede a destra si va verso elementi sempre più interni fino ad arrivare all'elemento rappresentato dall'espressione.

In Java le variabili d'istanza di ogni oggetto, ancor prima di essere esplicitamente inizializzate nel costruttore o in altri metodi, sono dal compilatore inizialmente poste automaticamente uguali a *zero* se sono di tipo numerico, al valore `false` se sono di tipo booleano e al valore `null` se sono delle referenze ad oggetti, dove **`null`** sta ad indicare che la referenza, pur dichiarata, non punta ancora ad alcun oggetto particolare.

Ma, oltre lle variabili d'istanza, ci possono essere anche **variabili di classe**, definite nella classe con la parola chiave **`static`**, che sono definite e memorizzate a livello di classe, così che tutti gli oggetti della classe creati ne condividono il valore, che è globale per tutte le istanze della classe e che, se un oggetto lo modifica, rimane modificato anche per tutti gli altri oggetti (vedremo più da vicino questo tipo di variabili nel paragrafo 2.1.9),

Lo specificatore **`final`**, posto davanti al nome di una variabile all'atto della sua dichiarazione e inizializzazione, indica che il suo valore è costante, nel senso che non può essere variato nel corso del programma (è come il `const` di C e C++). In Java viene usato il termine `final` al posto del `const` del C++ in quanto questa denominazione rientra in un concetto più generale nel linguaggio Java che chiama in questo modo variabili che non si possono cambiare di valore, metodi che non possono essere ridefiniti e classi che non possono essere derivate per eredità (subclassate), anche se questi tre concetti sono distinti. Un esempio può essere quello ripreso dalla classe `java.lang.Math`:

```
public static final double PI = 3.14159265358979323846;
```

Spesso le costanti di una classe sono definite come `static final`: non si vede infatti la ragione perchè esse, oltre a essere `final`, non siano anche definite come `static`, cioè esistenti in un solo esemplare condiviso da tutta la classe (non c'è proprio bisogno che ogni oggetto ne possieda una sua propria istanza).

Non solo una variabile di tipo elementare, ma anche un oggetto può essere dichiarato `final`, intendendosi con questo che non può essere modificato nelle sue variabili d'istanza. Per esempio:

```
final Color rosa = new Color(255, 220, 220);
```

In Java possono essere dichiarate `final` solo le variabili d'istanza o di classe, cioè quelle globali, non quelle locali, ossia quelle definite entro i metodi.

[Il termine **`threadsafe`** impone che la variabile non possa variare in modo asincrono.]

2.1.7 Metodi

2.1.7.1 Le funzioni in Java (metodi)

In Java, come si è detto, non esistono funzioni indipendenti come quelle del linguaggio C, accettate poi anche dal C++, o come le subroutine o le procedure di altri linguaggi, ma le funzioni, peraltro simili nel loro meccanismo alle funzioni dei linguaggi non OOP, sono definite solo all'interno delle classi come *funzioni membro* di esse e come tali sono più propriamente chiamate **metodi**, secondo la terminologia specifica dell'OOP.

I metodi sono sempre definiti all'interno del corpo della classe cui essi appartengono, e non è possibile definirli fuori, come si può fare invece nel C++ per le funzioni non *inline*.

2.1.7.2 Tipo di ritorno dei metodi

I metodi sono dichiarati facendo precedere al loro nome il **tipo**, cioè il tipo del risultato che essi restituiscono quando sono eseguiti, che può essere di un tipo base (`int`, `float`, etc.) o anche la referenza di un oggetto di una classe (in tal caso il tipo sarà espresso dal nome della classe), e che viene assegnato in un'istruzione `return` interna al metodo (vedi paragrafo 3.17.7) né più né meno di quello che avviene per le funzioni del C o di altri linguaggi. Si può adoperare un metodo in ogni posto di un programma dove possa comparire una variabile dello stesso tipo: in quel punto il metodo sarà eseguito ed al suo posto sarà messo il suo valore di ritorno. Ma un metodo può anche non restituire alcun risultato ma solo eseguire delle azioni (come le procedure o le subroutine di altri linguaggi) ed è allora dichiarato di tipo **`void`** (in tal caso il `return` entro il metodo non specifica alcun valore e indica solo un punto in cui il metodo si conclude e non è neanche sempre necessario).

Vediamo un esempio, con valore di ritorno di tipo elementare, di un metodo che restituisce il doppio del valore che riceve come argomento:

```
int doppio(int n) {
   return n * 2;
}
```

ed un altro con un valore di ritorno di tipo oggetto:

```
String str() {
   return "stringaDiRitorno";
}
```

di un metodo che restituisce un oggetto di tipo `String`:

Nel caso in cui il valore di ritorno di un metodo è di tipo array, nella definizione si può scrivere come nell'esempio seguente:

```
int[] metodo() { /* corpo del metodo */ }
```

ma è possibile anche scrivere la definizione nella forma:

```
int metodo()[] { /* corpo del metodo */ }
```

con le parentesi quadre caratteristiche dell'array che seguono la lista degli argomenti, anche se tale scrittura risulta meno leggibile della prima, che è più aderente alla forma generale.#

2.1.7.3 Argomenti di un metodo

Il nome di un metodo è sempre seguito da due parentesi **( )** che possono contenere o no degli argomenti. Nella definizione di un metodo gli **argomenti** (o **parametri**), che servono per trasmettere dei dati al metodo all'atto della sua chiamata, quando presenti, sono posti, in una determinata successione e separati tra di loro da virgole, tra le parentesi dopo il nome del metodo stesso, ciascuno con l'indicazione del proprio tipo e nome.. La lista degli argomenti di un metodo può essere considerata anche come una serie di dichiarazioni di variabili locali interne al metodo, il cui valore è inizializzato dall'esterno al momento dell'esecuzione del metodo con i valori assegnati loro entro parentesi nell'espressione che richiama il metodo.

Esempio di definizione di un metodo:

```
int metodo(int i, float f, String s) {
```

```
    /* corpo del metodo */
}
```

ed esempio di un'espressione che richiama quel metodo:

```
metodo(5, 2.2, "Ciao")
```

Quando un parametro di tipo elementare è passato ad un metodo non con un valore letterale ma con il nome di una variabile, come ad esempio:

```
int n = 2;
metodo(n, 2.2, "Ciao");
```

al metodo viene trasmessa una copia della variabile e non la variabile stessa (passaggio detto "per valore" o *by value*), perciò le variazioni subite all'interno del metodo da parte di questa copia (che nell'ambito del metodo funziona, secondo quanto già detto, come se fosse una variable locale) non si riflettono all'esterno del metodo sulla variabile originale. Esempio#:

Invece i parametri di tipo oggetto (inclusi array e stringhe) in Java sono passati ai metodi "per referenza" (*by reference*), e quindi possono essere variati (nel valore delle loro variabili d'istanza) all'interno del metodo cui sono stati passati.

2.1.7.4 Signature *di un metodo*

L'insieme del nome del metodo, del suo tipo di ritorno e del complesso dei suoi argomenti, ciascuno con il suo tipo (lista dei parametri) e disposti nel loro ordine, è chiamato con il termine inglese di ***signature*** del metodo e rappresenta una caratteristica importante per il riconoscimento del metodo quando esso viene chiamato.

2.1.7.5 Corpo del metodo

Il corpo del metodo è racchiuso da due parentesi `{` e `}` e contiene le istruzioni del metodo.

E' buona pratica di programmazione non scrivere metodi troppo lunghi per una loro migliore lettura e gestione.

2.1.7.6 Modificatori

La definizione di un metodo, così come la dichiarazione di una variabile d'istanza all'interno di una classe, può essere fatta precedere da uno o più modificatori. Anche le classi stesse quando sono definite possono essere precedute da un modificatore. Quando i modificatori sono più di uno l'ordine con cui sono scritti non è rilevante, anche se l'ordine generalmente seguito per i più comuni di essi è il seguente:

```
<public, protected o private>  static  abstract  synchronized  final  xxx#
```

Tra i modificatori ricordiamo innanzi tutto i qualificatori di accesso come `public`, `protected` o `private` che vedremo a fondo nel paragrafo 2.1.7.

Un metodo può essere dichiarato **final** e con questo gli viene imposto di non potere essere ridefinito in una sottoclasse derivata per eredità dalla classe in cui è definito e di rimanere quindi immutato in tutte le classi discendenti che lo ereditano.

Un metodo **abstract** è un metodo definito soltanto a livello di prototipo nella sua *signature* e non ancora implementato, il cui corpo cioè non è ancora definito. Una classe contenente metodi astratti non può che essere anch'essa astratta e quindi non istanziabile direttamente (vedi paragrafo 2.1.11).

Un metodo è preceduto dal termine **native** se è stato scritto in un altro linguaggio (nativo, di solito C), e da **synchronized** se esso è multithreaded e c'è un blocco prima della sua esecuzione.

Il modificatore **transient** viene applicato a quelle variabili che non sono persistenti nelle operazioni che producono una persistenza (conservazione dello stato) all'oggetto.

2.1.7.7 Metodo costruttore

Tra i metodi di ogni classe ce n'è uno particolare che porta lo stesso nome della classe e che non restituisce alcun valore e non è indicato neanche come `void`: si tratta del metodo detto **costruttore** (*constructor*), che viene eseguito automaticamente al momento della creazione di un oggetto con il `new`, cioè all'atto dell'istanziazione della classe in un oggetto. Il costruttore in generale viene definito ed usato per svolgere delle operazioni di inizializzazione dell'oggetto stesso al momento della sua creazione, ma nulla vieta al programmatore di farne anche un altro uso inserendo in esso altre operazioni (nel caso particolare di un'applet operazioni di inizializzazione possono essere generalmente svolte anche, come vedremo, in un metodo particolare dell'applet chiamato `init()` proprio della classe `Applet`). Se un costruttore è definito `public` (vedi paragrafo 2.1.7) la classe diviene disponibile per la creazione di oggetti anche al di fuori del package in cui è definita. Esempio:#

Anche il costruttore, come ogni altro metodo, gode della facoltà di *overloading*, e pertanto di esso ci possono essere contemporaneamente varie versioni coesistenti nell'ambito della stessa classe definite in modi diversi, con argomenti differenti in numero e tipo in ciascuna di esse (diverse *signature*), in corrispondenza di diversi modi possibili di inizializzare gli oggetti della classe.

Il costruttore si distingue dagli altri metodi della classe per vari aspetti:

- ha sempre lo stesso nome della classe,

- non presenta un tipo di ritorno (si potrebbe dire che il suo valore di ritorno è l'oggetto stesso creato),

- è richiamato automaticamente dall'operatore `new` quando si crea l'oggetto,

- il suo modificatore di accesso specifica la possibilità o meno che l'oggetto sia creato...

- non può essere chiamato direttamente da altri oggetti, ma solo in certe circostanze dall'oggetto stesso (vedi paragrafo 2.1.13) o da oggetti di classi derivate (vedi paragrafo 2.1.14),

Se in una classe non è stato definito esplicitamente alcun costruttore il compilatore provvede a fornirne automaticamente uno di default avente come sola istruzione la chiamata del costruttore della sua superclasse, cioè della classe da cui la classe stessa deriva, avendo sempre ogni classe una sua classe genitrice (vedasi paragrafo 2.1.11) e, in mancanza, di quella ancora superiore e così via su su nella scala di discendenza della classe.

Può accadere che, una volta che si sia definito un costruttore, se ne voglia poi definire un secondo, che, con una lista di parametri diversa, faccia le stesse cose che fa il primo, aggiungendovi però qualcosa di particolare: in tal caso questo secondo costruttore può richiamare il primo (e lo potrà fare, come vedremo con il metodo `this()`) e poi aggiungervi le sue istruzioni particolari. Esempio:#

```
// Esempio 2.x
// costr.java   F.Spagna
class miaClasse {
   c() {}
   c() {
      this();
   }
}
```

Le classi Java non hanno un metodo *destructor* come quello delle classi in C++, perchè se ne sente meno la necessità in quanto in Java è il compilatore stesso che si incarica della liberazione della memoria occupata da un oggetto quando questo cessa di essere utilizzato, ed era proprio questa generalmente la principale attività di ogni destructor (chi non proviene dal C++ tralasci pure questi riferimenti alle differenze tra i due linguaggi che vengono fatti di tanto in tanto, che possono essere utili ai programmatori provenienti da quei linguaggi, ma che non sono affatto necessari agli altri).

2.1.7.8 Overloading *dei metodi*

In Java diversi metodi della stessa classe possono avere lo stesso nome, purchè sia diversa la lista (in numero o tipo) degli argomenti (abbiano cioè *signature* diverse): questa caratteristica è chiamata **overloading dei metodi** e contribuisce a fornire al linguaggio la proprietà del **polimorfismo**, un aspetto molto importante nella programmazione ad oggetti.

Quando il compilatore incontra in un programma una chiamata ad un metodo di un oggetto *overloaded* o polimorfico, cioè esistente sotto lo stesso nome in versioni multiple (con diverse *signature* e definizioni diverse), va a confrontare i metodi della classe dell'oggetto portanti lo stesso nome, in base al tipo e al numero dei parametri che gli vengono passati, e se ne trova uno che risponde alla *signature* della chiamata sceglie quello per utilizzarlo, ma se non ne trova nessuno che corrisponde, allora segnala un errore.

L'uso di metodi polimorfici permette di avere un'interfaccia più semplice degli oggetti verso l'esterno (cioè verso altri oggetti) quando si ha a che fare con diversi modi per fare lo stesso tipo concettuale di operazione. Tipico è ad esempio il caso di *overloading* dei costruttori di una classe (vedi paragrafo precedente): poiché un costruttore ha il nome obbligato (che coincide con quello stesso della classe), quando per costruire un oggetto ci sono modi diversi (con differenti tipi di parametri) l'*overloading* si rivela necessario (vedi gli esempi fatti nel paragrafo 2.1.2 per la classe `Scritta` e la classe `Punto` ciascuna delle quali prevede due costruttori polimorfici).

Si noti che non si può fare l'*overloading* di due metodi che abbiano lo stesso numero e tipo di argomenti ma differiscano solo per il tipo di ritorno.

2.1.7.9 Invocazione di un metodo

Nella programmazione ad oggetti (OOP) l'invocazione di un metodo di un oggetto viene considerato come l'*invio di un messaggio all'oggetto.*

Come le variabili d'istanza, anche i metodi di un oggetto di una classe possono essere richiamati con il loro semplice nome dai metodi della classe stessa, per i quali essi sono sempre

accessibili. Per poter invece essere richiamati dai metodi di un altro oggetto devono essere accessibili all'esterno, e questo succede se sono chiamati da metodi delle classi dello stesso package o sono stati dichiarati `public`, e il loro nome deve essere preceduto da quello dell'oggetto unito con l'operatore punto (**.**) interposto tra il nome dell'oggetto a sinistra e quello del metodo a destra. Così il metodo `scrivi()` dell'oggetto `scr` della classe `B01Scritta` dichiarato `public` nel paragrafo 2.1.3 può essere richiamato con l'espressione `scr.scrivi(20, 30)` se `20` e `30` sono per esempio i parametri passati al metodo, cioè le coordinate della posizione della scritta.

Se un oggetto `ogg` ha un metodo `met()` che restituisce un altro oggetto che abbia una variabile d'istanza `var` questa può essere richiamata come

```
ogg.met().var
```

che rappresenta la variabile d'istanza `var` dell'oggetto `ogg.met()`.

La chiamata di un metodo `public` di un oggetto può anche essere fatta contestualmente alla creazione dell'oggetto stesso con un'unica espressione del tipo:

```
new B01Scritta(" ").scrivi(30, 40);
```

ancora con riferimento all'esempio del paragrafo 2.1.2. In tal modo però l'oggetto viene creato e usato solo per il suo metodo, ma non ne viene creata una referenza che possa poi essere utilizzata nel programma. Esempio:#

Java ammette la **ricorsività** dei metodi, in quanto essi possono al loro interno richiamare se stessi. Esempio:#

2.1.8 Accesso ad una classe e modificatori delle classi

Il termine posto davanti al nome della classe del paragrafo 2.1.2 è un ***modificatore***, che, essendo in particolare **`public`**, rende la classe accessibile da parte di oggetti anche esterni al package; se invece la classe non fosse dichiarata `public` essa sarebbe riconosciuta soltanto nell'ambito del package nella quale è dichiarata e non sarebbe accessibile dai metodi di altre classi ma solo di quelle dello stesso package.

Varie classi possono essere definite nello stesso package, ma una sola può essere `public`, le altre sono `private` per default o per espressa dichiarazione `private`.

2.1.9 Accesso a variabili d'istanza e metodi di un oggetto

L'accesso dall'esterno ai membri (variabili d'istanza e metodi) di una classe è regolato da **modificatori di accesso** che vengono applicati ad essi e sono costituiti da parole chiave che specificano il livello di accesso (o, se si vuole, la *visibilità* o *portata*) di variabili e metodi dall'interno di altre classi esterne alla classe stessa. Un metodo di una certa classe che sia visibile da un'altra classe può essere invocato dagli oggetti di essa ed operare su questi ed una variabile di una certa classe visibile da un'altra classe può essere usata e modificata dagli oggetti di quella. Un modificatore di accesso, come qualunque altro modificatore, è posto davanti alla specificazione di tipo all'atto della dichiarazione (di solito, ma non necessariamente, in prima posizione rispetto ad eventuali altri modificatori), come per esempio:

```
public int var;
public void metodo() { }
```

Tali modificatori di accesso possono essere specificati con diversi livelli di protezione in uno dei modi elencati di seguito, partendo dalla minima protezione verso visibilità sempre più ristrette (siccome per caso le quattro parole che definiscono tali modalità di accesso cominciano tutte con una lettera P si parla dei "4 P": `public`, `package`, `protected` e `private`):

- **`public`** rende variabili e metodi liberamente accessibili, oltre che dai metodi interni dell'oggetto stesso, anche da quelli di oggetti di qualsiasi altra classe anche esterna al package di cui fa parte la classe: è questo il livello di massima visibilità; in particolare il costruttore di una classe deve essere `public` perchè si possano creare oggetti della classe dall'interno di altre classi, in quanto la creazione di ogni nuovo oggetto richiama tale metodo.

- La caratteristica di **default**, cioè quella che si ha nel caso che non si specifichi nulla riguardo all'accesso, è quella secondo la quale si assume che variabili d'istanza e metodi siano accessibili ***all'interno del package*** che li contiene e solo in esso (questo livello di accessibilità, che potremmo chiamare **visibilità di package**, è chiamata da qualcuno anche "*friendly scope*"); questo è il livello di visibilità di default perché si presume che le classi di uno stesso package generalmente sono legate tra di loro più che con l'esterno in quanto condividono uno stesso contesto. Il package rappresenta un nuovo strato di *scope* (visibilità) rispetto al C++, in quanto la visibilità di classi, variabili e metodi in esso contenuti è per default totale nell'ambito del package stesso, salvo che non sia specificato diversamente, come si vedrà al paragrafo 2.1.7 per variabili e metodi.

- **`protected`** indica che variabili e metodi sono, come quelli `private`, accessibili soltanto dai metodi della classe stessa e non dall'esterno, ma, a differenza di quelli `private`, sono ereditabili dalle sottoclassi# (sono cioè presenti, e riconosciuti come `protected`, anche nelle sottoclassi derivate). In generale si definisce un metodo come `protected` quando, prevedendo che la classe possa essere prima o poi estesa per eredità, si vuole che esso sia privato rispetto ad altre classi, ma nello stesso tempo ereditabile nelle sottoclassi.

- **`private`** indica invece che l'accessibilità è ristretta solo ***all'interno dell'oggetto stesso***, e cioè possono essere richiamati soltanto da un altro metodo dello stesso oggetto e non da altri oggetti: le variabili `private` non possono quindi essere viste o modificate da altri oggetti esterni a quello cui esse appartengono e sono così protette: si dice allora che i dati sono ***incapsulati*** (dati utilizzati solo per il funzionamento interno della classe ma protetti da lettura e scrittura verso l'esterno). I membri `private` servono in generale per il funzionamento interno della classe senza che sia necessario che siano conosciuti da fuori, così restano sicuri e nascosti, e gli utilizzatori della classe non hanno perfino neanche bisogno di conoscerne l'esistenza. Le variabili d'istanza ed i metodi `private` di una classe non sono ereditati dalle sottoclassi derivate per estensione da essa, e per di più in una sottoclasse non ne possono essere definiti dei nuovi con lo stesso nome di variabili e metodi `private` esistenti nella superclasse, e cioè non solo non possono essere ereditati, ma neanche ridefiniti nelle loro sottoclassi.

Le variabili d'istanza di una classe possono essere considerate come ***globali*** per la classe stessa nel senso che all'interno di essa sono viste (accessibili) dappertutto, cioè all'interno di tutti i metodi, e pertanto possono essere modificate da qualunque di essi, mentre invece le variabili create all'interno dei metodi, così come quelle definite come argomenti del metodo, sono considerate ***locali*** poiché sono viste solo all'interno dello stesso metodo nel quale sono state create e possono essere lette o modificate (accesso in lettura o scrittura) solo nell'ambito di esso.

Metodi e variabili di una classe devono essere dichiarati `public` e quindi esposti all'esterno solo a ragion veduta perché uno dei compiti principali di un oggetto è proprio quello

di incapsulare, e cioè nascondere all'esterno, i suoi dati in modo da non esporli a manipolazioni incontrollate (anche se si volesse aprire alla sola lettura le variabili le si esporrebbero però anche a modifica perché lettura e scrittura dal punto di vista dell'accesso non sono distinte).

In generale sono dichiarati `public` le variabili ed i metodi che si vuole siano interfacciati con l'esterno, come è tipicamente per i metodi di accesso per leggere o scrivere (modificare) il valore delle variabili d'istanza delle classi.

2.1.10 Visibilità (*scope*) delle variabili

Quando Java incontra il nome di una variabile entro un certo metodo, cerca dapprima se una variabile con quel nome è stata dichiarata nell'eventuale blocco corrente (cioè quella parte di codice racchiuso da due parentesi { e }); se non ve ne trova una con quel nome in quel blocco la cerca nei blocchi via via più esterni (se si è in presenza di diversi blocchi uno annidato dentro l'altro) fino ad arrivare al blocco di definizione del metodo stesso, ed usa la più interna che trova; se in tutto il metodo non trova una dichiarazione di quella variabile, allora la cerca come variabile d'istanza della classe. Se non ne trova neanche una come tale va a vedere se è stata ereditata da altre classi risalendo via via tutta la gerarchia delle superclassi in questa ricerca per fermarsi alla prima dichiarazione incontrata.

Se una variabile è definita ad un certo livello di *scope*, quando ad un livello superiore ce n'è già un'altra con lo stesso nome quella superiore (o più esterna) rimane "nascosta" da quella a livello inferiore. Ma è meglio evitare che questo accada, perché può essere fonte di errori. Le cose poi si complicano quando a livello di eredità nelle sottoclassi si introducono delle variabili d'istanza con lo stesso nome di quelle delle superclassi e in tal caso gli errori diventano a volte difficili da trovare.

Vediamo un esempio (#di pagina 102, da ampliare con il this) in cui esistono a vari livelli variabili con lo stesso nome:

```
// B04miaClasse (F.Spagna) Variabili con lo stesso nome

class B04miaClasse {

   int a = 1;                    // variabile d'istanza (globale alla classe)

   public static void main(String arg[]) {

   int a = 2; // variabile locale al metodo (nasconde quella d'istanza)
      System.out.println("locale: " + a);                  // stampa 2
      System.out.println("variabile d'istanza: " + this.a);    // stampa 1
   }
}
```

Il risultato è:

#

Nell'esempio si vede anche come tra una variabile locale ed una variabile d'istanza l'ambiguità dovuta ad omonimia può essere risolta ricorrendo al `this` (vedi paragrafo 2.1.13).

2.1.11 Variabili e metodi `static` (di classe)

Il termine **static** applicato ad una variabile membro di una classe indica che della variabile esiste una sola istanza (un esemplare) condivisa da tutti gli oggetti che si vengano a creare della classe stessa, quindi le variabili `static` hanno la natura di **variabili globali** con valori generali per tutta la classe e non particolari per ciascuna sua particolare istanza (oggetto) come è per le variabili d'istanza: sono quelle che si chiamano **variabili di classe**.

Poichè le **variabili statiche o di classe** conservano un valore globale a tutta la classe, cioè per tutte le istanze di essa, se il loro valore viene modificato in una qualsiasi istanza che le stia adoperando, risulta di conseguenza modificato anche per tutti gli altri oggetti della classe, che lo condividono. Tipicamente le variabili statiche sono delle costanti usate dalla classe, ma variabili statiche potrebbero, volendo, anche essere usate per far comunicare attraverso di esse diversi oggetti della stessa classe (possiamo immaginare ad esempio una variabile contatore che conta certe operazioni fatte da qualsiasi oggetto della classe).

Le variabili statiche di un oggetto di una classe possono essere inizializzate, oltre che come quelle di istanza non statiche, cioè al momento della dichiarazione o nell'ambito di un metodo, anche mediante un blocco di codice (racchiuso da due parentesi `{` e `}`), definito anch'esso `static`, posto nel corpo della classe eccezionalmente al di fuori di ogni metodo, come è mostrato nell'esempio seguente:

```
class miaClasse {
   static int x[];                          // variabile di classe
   static {                                 // blocco statico di codice
      for (int n = 0; n < 10; n++) x[n] = 1;
   }
   // resto del corpo della classe
}
```

Le costanti definite nell'ambito di una classe sono in generale dichiarate come `final static`: sono `final` in quanto costanti nel senso che non vengono mai cambiate nel corso del programma e `static` in quanto ce n'è una sola istanza per tutta la classe (non è che ogni oggetto ne memorizzi una sua propria e questo è anche quello che ci si aspetta da una costante generale).

Come si possono definire delle **variabili di classe**, così possono anche esserci dei **metodi di classe** definiti come `static` . I **metodi statici** di una classe possono agire soltanto su variabili statiche e metodi statici della classe stessa e non possono adoperare i membri non statici che sarebbero propri di ogni particolare istanza della classe (in particolare essi non possono adoperare la variabile `this` che è riferita ad un oggetto specifico, cioè quello in cui si lavora, vedi paragrafo 2.1.13). Metodi e variabili statiche restano confinati in un dominio della classe non coinvolto con istanze particolari di essa e costituiscono un insieme di caratteristiche e funzionalità generali proprie della classe che non operano sui suoi singoli oggetti, anche se possono essere usati da essi. Si osservi che le variabili locali definite all'interno di un metodo statico non è necessario che siano dichiarate statiche perché tutto nell'ambito del metodo è già in un contesto statico.

I membri (variabili e metodi) `static` di un oggetto di una classe possono essere adoperati anche se non è stato istanziato alcun oggetto della classe. Tali membri, naturalmente se sono anche `public` e quindi accessibili, possono essere richiamati dai metodi di oggetti di altre classi nella loro qualità di membri della classe senza che se ne debba prima istanziare un particolare oggetto, ma il linguaggio tollera la chiamata di membri statici come membri di un oggetto creato di quella classe: anche se questo è tollerato è per ragioni di chiarezza del codice che è però comunque meglio in tali casi usare il nome della classe piuttosto che quello dell'oggetto (l'oggetto specifico ha poco a che fare con un membro statico).

Ad esempio, se una classe `miaClas` è definita come:

```
class miaClas {
   static int v;
   static void met() { /* . . . */ }
}
```

e se ne dichiara un oggetto `mioOgg`:

```
miaClas mioOgg;
```

la variabile d'istanza statica `v` ed il metodo statico `met()` possono essere richiamati come:

`miaClas.v` e `miaClas.met()` (cioè come appartenenti alla classe)

oppure (e ciò è solo tollerato) come:

`mioOgg.v` e `mioOgg.met()` (cioè come appartenenti all'oggetto)

Ecco perchè si possono usare nel codice Java espressioni che non fanno riferimento ad oggetti particolari, come:

```
System.exit(0);
```

espressione molto frequente usata per l'uscita da un programma, oppure quella usatissima:

```
System.out.println();
```

per scrivere sull'output standard del sistema mediante il metodo `println()` del membro `out` di `System`, dove `System` è il nome di una classe e non di un oggetto. Infatti la classe `System` del package `java.lang` è una classe che contiene i comportamenti specifici del sistema ed ha tutte le variabili ed i metodi definiti come `static` (ed ha per di più un costruttore `private`, che comporta che di di essa non si possa proprio creare alcun oggetto).

2.1.12 Classi `static` non istanziabili

Si è visto al paragrafo precedente come i membri di una classe possono essere `static`, ma anche una classe stessa può essere dichiarata **`static`**, significando questo che essa non può essere istanziata. Nei due casi di esempi riportati alla fine del paragrafo precedente, oltre che essere in presenza di membri statici, non si era detto che la classe stessa (la `System`) era `static`.

Un'altra tipica classe `static` è la `Math` (anch'essa del package `java.lang`), che fornisce tutte le funzioni matematiche definite come particolari metodi della classe. Per esempio `Math.cos()` fornisce la funzione coseno di un numero come metodo della classe.

Questi casi ci aiutano a capire il concetto di classe statica: non si vede infatti a cosa potrebbe servire avere la possibilità di istanziare dei particolari oggetti di una classe generale come la `Math` il cui compito è solo quello di fornire delle funzioni matematiche.

2.1.13 Estensione o eredità di una classe da un'altra

In Java si può creare una nuova classe derivandola per **estensione**, che è il termine usato in Java per l'**eredità**, da un'altra classe già definita, che rappresenta la sua **superclasse** (si potrebbe anche chiamare *classe base*): la nuova classe (che è detta **sottoclasse** della classe originaria da cui è stata derivata) ha la stessa struttura e sa fare tutto quello che fa la sua superclasse, in quanto ne eredita tutte le caratteristiche e funzionalità, cioè tutte le variabili d'istanza ed i metodi `public` e `protected` (ma non quelli `private`), ma in più nella classe derivata si possono aggiungere altre variabili o nuovi metodi, specifici per i compiti che si vogliono assegnare alla nuova classe che si sta definendo o si possono ridefinire metodi già esistenti nella superclasse.

L'operazione di creazione di una nuova classe come derivata da un'altra per eredità è comunemente detta ***subclassing*** (che in inglese vuol dire "fare una sottoclasse"). La nuova classe può a sua volta dare origine ad altre sue sottoclassi, e così via, venendosi così a creare complessivamente una **gerarchia di classi**.

In generale l'eredità nei linguaggi OOP viene utilizzata principalmente per due scopi:

- aggiungere o cambiare qualcosa, a livello delle caratteristiche (dati) o funzionalità (metodi), in classi già disponibili per adattarle o estenderle a nuove proprietà o compiti specifici aggiuntivi, passando generalmente da comportamenti più generali a comportamenti più specializzati;

- creare un complesso di classi organico e strutturato gerarchicamente a partire da una classe radice con comportamenti via via più particolari scendendo nella gerarchia: questo richiede una pianificazione e la stessa libreria di classi di Java è organizzata in questo modo; una strutturazione di classi secondo questo criterio presenta anche il vantaggio, nel caso che una classe di un livello superiore nella gerarchia ad un certo momento venisse per qualche ragione cambiata: l'eredità fa sì che in tal caso si deve sì ricompilare la nuova classe, ma non è necessario ricompilare anche tutte le altre classi discendenti da essa.

Per fare l'estensione, nella definizione della sottoclasse dopo il suo nome si scrive il nome della superclasse preceduto dalla parola chiave **`extends`**:

```
class nuovaClasse extends classeEsistente {
   // corpo della nuova classe
   // contenente solo le variabili e i metodi aggiuntivi
}
```

Riprendendo la classse `Scritta` definita nell'esempio del paragrafo 2.1.2 si potrebbe derivarne una nuova classe `scrittaColorata` che avesse le stesse proprietà di quella, ma fosse arricchita di una nuova variabile d'istanza (`int colore`) rappresentante il colore della scritta e di un nuovo metodo (`void colora()`) che assegnasse alla scritta un dato colore:

```
// B05scrittaColorata.java   F.Spagna   Esempio di eredita'

class B05scrittaColorata extends Scritta {
   int colore;                              // nuova variabile d'istanza
   public void colora(int r, int g, int b) {          // nuovo metodo
      // istruzioni del metodo per dare un colore alla Scritta
   }
}
```

Si noti che nell'estensione di una classe in un'altra non solo non è richiesta la riscrittura esplicita del codice della classe da cui si fa l'estensione (superclasse), ma non si ha neanche il bisogno di sapere come è fatta internamente la superclasse.

L'oggetto di una classe derivata da un'altra contiene l'insieme di tutte le variabili d'istanza e di tutti i metodi definiti via via in tutti i componenti della sua linea ascendente di eredità, dalla sua superclasse diretta andando in su nella gerarchia fino alla classe `Object`. Se si risale infatti in ogni gerarchia di classi derivate, in Java tutte le classi sono in ultimo originate da una stessa classe chiamata **`Object`** (contenuta nel package `java.lang`) che è la capostipite (radice) di tutte le classi e fornisce a tutti gli oggetti Java un comportamento di base comune. Se si dichiara una nuova classe senza farla derivare esplicitamente da un'altra, essa è assunta automaticamente essere una sottoclasse diretta della classe `Object`. Quindi ogni classe in Java ha sempre almeno le proprietà della classe `Object`. Per esempio, poichè la classe `Object` ha un metodo `Class getClass()`, che restituisce la classe dell'oggetto, un metodo `String toString()`, che restituisce una stringa rappresentante il nome della classe cui appartiene un oggetto, e un metodo `boolean equals(Object obj)` che verifica l'uguaglianza di un oggetto con un altro qualsiasi, ogni classe, e quindi ogni oggetto, in Java possiede sempre almeno questi metodi. È raccomandato che per ogni nuova classe si ridefinisca sempre il metodo `toString()` proprio della nuova classe.

Esempio con metodi derivati da Object#:

```
class B06nuovaClasse {
   // c
}
```

Se una classe è dichiarata **`final`** non se ne possono derivare per estensione delle sottoclassi. Questa proprietà, che viene utilizzata la maggior parte delle volte per ragioni di sicurezza, permette comunque al compilatore di fare una certa ottimizzazione. Tra le classi di sistema di Java ce ne sono diverse, come la classe `String`, che sono dichiarate `final`, e che quindi non possono dare origine a sottoclassi.

Una **classe astratta**, che è definita come **`abstract`**, é una classe non implementata (cioè non completamente definita) e perciò non direttamente istanziabile: infatti essa contiene uno o più metodi `abstract`, che sono metodi dichiarati ma non definiti (vedi paragrafo 2.1.6.6), la cui implementazione viene fatta soltanto nelle sottoclassi da essa derivate, che solo allora possono essere istanziate (`abstract` è in tal senso simile al `virtual` delle classi del C++). Le classi di questo tipo sono predisposte al solo scopo di funzionare come classi base da cui si intende creare una discendenza, ma non sono utilizzabili direttamente: se ne prevede solo la struttura ed il comportamento generale, lasciando la definizione dei comportamenti particolari alle classi derivate da esse. In questo senso le classi `abstract` sono un po' l'opposto delle classi `final` che non possono invece essere derivate.

Il Java, contrariamente ai linguaggi C++ e a Smalltalk, che permettono invece l'eredità multipla cioè contemporanea da più classi, ammette soltanto l'**eredità singola** cioè da una sola classe, in altre parole una classe può avere una sola superclasse. Quando questo viene a costituire una limitazione per la programmazione (perché l'eredità multipla offre un mezzo potente in OOP, anche se complica la programmazione: ecco perché, pur con qualche rimpianto, vi si è rinunziato in Java), risultati simili a quelli ottenibili con l'eredità multipla possono essere ottenuti in Java con l'implementazione di interfacce da parte di una classe, come vedremo in seguito.

2.1.14 Variabile `this` e metodo `this()`

Ogni oggetto ha in Java a disposizione la variabile speciale **`this`** che può essere usata all'interno dei suoi metodi, ma solo, e se ne capirà subito la ragione, da quelli non statici. Questa variabile rappresenta l'oggetto stesso, e più precisamente la referenza ad esso, ad esempio le espressioni `this.v` e `this.met()` rappresentano rispettivamente la variabile d'istanza `v` ed il metodo `met()` dell'oggetto stesso nell'ambito del quale sono utilizzati. Tale variabile può essere usata ad esempio quando c'è ambiguità, come nel caso in cui una variabile locale o un argomento di un metodo abbia lo stesso nome di una variabile d'istanza dell'oggetto (vedi esempio sotto), oppure quando un metodo richiede l'oggetto stesso come argomento o come valore di un `return`.

Un esempio tipico di utilizzazione della variabile `this` può essere il seguente, che si richiama all'esempio del paragrafo 2.1.2 in cui il costruttore questa volta è scritto esattamente come lo è nella versione originale della classe `java.awt.Point` di sistema di Java:

```
public class Punto {
   public int x;
   public int y;

   public Punto(int x, int y) {
      x = this.x;
      y = this.y;
   }
}
```

In questo caso il `this` serve per distinguere le variabili d'istanza globali `x` e `y` della classe da quelle locali (in questo caso gli argomenti) del metodo costruttore, che hanno lo stesso nome. In generale il `this` può essere utilizzato per raggiungere una variabile d'istanza "nascosta" da una variabile locale. Esempio#:

Si noti che quando nell'ambito di un metodo di un oggetto vengono chiamati una variabile d'istanza `var` o un altro metodo `met()` dello stesso oggetto con il loro semplice nome è come se si usasse implicitamente il `this`, e cioè `this.var` o `this.met()`.

Un esempio invece in cui si passa ad un metodo come argomento l'oggetto stesso può essere il seguente:

```
public class Punto {
   fare l'esempio;
}
```

Poiché il `this` si riferisce ad una particolare istanza di una classe (l'oggetto stesso in cui esso viene usato) si capisce perché esso non può essere usato in metodi di classe (statici).

Per richiamare il costruttore di un oggetto da un metodo dell'oggetto stesso si può usare il metodo **`this()`**. Esempio#:

2.1.15 Variabile `super` e metodo `super()`

Per ogni oggetto la variabile **`super`** rappresenta la superclasse dell'oggetto stesso. Tale variabile può essere adoperata quando, per qualche ragione, si voglia utilizzare, anzichè un

metodo ridefinito per *overriding* a livello sottoclasse, il metodo nella forma originaria della sua superclasse, che sarà richiamato con l'espressione:

```
super.nomeMetodo(parametri);
```

Ma se in particolare si vuole utilizzare il metodo costruttore della superclasse si ha a disposizione in una classe derivata il metodo **`super()`** che rappresenta appunto il costruttore della superclasse:

```
super(parametri);
```

Un esempio è stato visto al paragrafo 2.1.13 a proposito del costruttore di sottoclassi.

Nel costruttore di una sottoclasse si può richiamare (necessariamente come prima istruzione) il costruttore della sua superclasse quando se ne voglia utilizzare il contenuto anche nella sottoclasse:

```
super(parametri);
```

e di seguito ad una tale istruzione, se è il caso, possono essere poi aggiunte altre nuove istruzioni particolari della sottoclasse stessa, inesistenti nel costruttore della superclasse: in tal modo si evita di riscrivere completamente il nuovo costruttore, ma se ne scrivono solo le modifiche. Un tale procedimento va a vantaggio della sicurezza perché si può anche ignorare tutto quello che contiene il costruttore della superclasse.

2.1.16 Ridefinizione (*overriding*) dei metodi ereditati

In una classe derivata si può anche ridefinire un metodo ereditato dalla sua superclasse, per adattarlo a nuove esigenze, lasciandogli lo stesso nome e gli stessi parametri (stessa *signature* di quella del metodo della superclasse): tale operazione è detta in inglese ***overriding***. Questa possibilità contribuisce, insieme all'overloading dei metodi, di cui si è già detto al paragrafo 2.1.6.8, al **polimorfismo** delle classi, proprietà molto importante dell'OOP. Però in una sottoclasse non si possono cambiare le caratteristiche di accesso (da `public` a `private` o viceversa) di variabili d'istanza o di metodi ereditati dalla superclasse.

Esempio di overriding di un metodo delle superclasse in una classe derivata:

```
class animale {
   cheAnimale() {
      System.out.println("sono un animale");
   }
}

class cane extends animale {
   cheAnimale() {
      System.out.println("sono un cane");
   }
}
```

Si può anche fare un *overriding* di un metodo di una superclasse con un metodo nella sottoclasse avente parametri diversi (*oveloading* di un metodo ereditato).

Può presentarsi il caso che, quando in una nuova classe si fa l'*overriding* di un metodo di una sua superclasse, non si voglia per qualche ragione riscrivere completamente il codice del

metodo della superclasse, ma lo si voglia conservare nella sua forma originaria per poi solo aggiungervi qualche cosa di particolare alla sottoclasse. In tal caso nel nuovo metodo `met()` si può chiamare il metodo della superclasse con:

```
super.met();
```

e poi aggiungervi altre nuove istruzioni.

Se ne fa qui un altro esempio riprendendo quello precedente:

```
class animale {
   cheAnimale() {
      System.out.println("sono un animale");
   }
}

class cane extends animale {
   cheAnimale() {
      super();
      System.out.println(" e in particolare un cane");
   }
}
```

2.2 Polimorfismo delle classi

2.2.1 Referenze a classi e sottoclassi

Si è visto come ad una variabile oggetto sia possibile assegnare la referenza di un oggetto creato dello stesso tipo:

```
miaClasse mc = new miaClasse();
```

ma è anche possibile assegnare ad una referenza oggetto di una certa classe la referenza di un oggetto di una sua sottoclasse. Ad esempio se la classe:

```
class animale {}
```

è una superclasse che dà origine per eredità alla sottoclasse:

```
class cane extends animale {}
```

si può fare un'assegnazione del tipo:

```
cane Fido = new cane();
```

ma anche:

```
animale Fido = new cane();
```

ed è anche possibile passare un parametro del tipo della sottoclasse ad un metodo che si aspetta invece come argomento una variabile di tipo della superclasse, ad esempio ad un metodo:

```
metodo(animale a);
```

si può passare il parametro:

```
metodo(Fido);
```

Così nel posto in cui in un programma è attesa la referenza ad un oggetto di una certa classe si può porre la referenza ad un oggetto di una sua sottoclasse e ciò è possibile perché una sottoclasse contiene già in sé tutte le informazioni richieste dalla sua superclasse.

Esempio (argomento inizializzato#):

Questo anche se la superclasse fosse astratta e quindi non istanziabile direttamente e se invece la sottoclasse è istanziabile.

L'inverso, e cioè l'uso di referenze di una superclasse dove ci si aspetta una referenza di una sua sottoclasse, non può essere fatto se non creando un nuovo oggetto nella classe voluta mediante un'operazione di casting (vedi paragrafo 2.3).

Quando viene invocato un metodo di un oggetto di una certa classe, Java adotta il metodo che abbia la stessa firma (*signature*), cioè lo stesso tipo di ritorno e la stessa lista (numero e tipo) degli argomenti, eventualmente definito nella classe stessa o, altrimenti, se in essa non ne trova uno, va a cercarlo nella sua superclasse e adotta il primo con la stessa signature che incontra risalendo via via nella gerarchia di eredità della classe.

Se in una classe è definito (per *overriding*) un metodo avente la stessa *signature* di uno definito in una sua superclasse quello della superclasse rimane "nascosto" da quello della sottoclasse (può essere però raggiunto, come vedremo, con il `super`). Esempio#:

2.2.2 Il polimorfismo

Ammettiamo di aver definito una classe di base chiamata `musicista` avente un metodo `suona()`:

```
class musicista {
   void suona() { }
}
```

classe che può anche essere astratta, se il metodo non è definito (è cioè sprovvisto di un corpo), ed in tal caso non istanziabile direttamente:

```
abstract class musicista {
   void suona();
}
```

e poi di aver specializzato la classe in diverse sue sottoclassi, in ciascuna delle quali è definita una propria particolare versione del metodo `suona()`:

```
class violino extends musicista {
   void suona() {
      System.out.println("suona il violino");
   }
}
class violoncello extends musicista {
   void suona() {
      System.out.println("suona il violoncello");
   }
}
```

```
class pianoforte extends musicista {
   void suona() {
      System.out.println("suona il pianoforte");
   }
}
```

E' allora possibile, come si è detto al paragrafo precedente, referenziare delle istanze della superclasse `musicista` con oggetti delle classi derivate:

```
musicista Oistrack = new violino();
musicista Rostropovich = new violoncello();
musicista Oborine = new pianoforte();
```

Quando si invocano i metodi:

```
Oistrack.suona();           // suona il violino
Rostropovich.suona ();      // suona il violoncello
Oborine.suona();            // suona il pianoforte
```

pur essendo i tre oggetti tutti della classe `musicista` ed essendo per essi stato chiamato un metodo avente sempre lo stesso nome, ognuno farà agire la sua particolare versione del metodo `suona()` (potremmo dire, con un'immagine, che ognuno dei musicisti suonerà il suo proprio strumento). Questo è un esempio di applicazione della proprietà del **polimorfismo** delle classi, secondo la quale gli stessi metodi di diversi oggetti della stessa classe possono presentare dei comportamenti diversi a seconda dell'oggetto.

Per rafforzare il concetto consideriamo una variante del caso precedente definendo un array di musicisti (insieme dei componenti del trio):

```
musicista trio[] = new musicista[3];

trio[0] = new violino();
trio[1] = new violoncello();
trio[2] = new pianoforte();
```

e farli suonare tutti insieme con un unico comando (istruzione):

```
for (int n = 0; n < 3; n++)
   trio[n].suona();
```

dove il vantaggio di poter usare metodi con lo stesso nome risulta ancor più evidente.

Ecco il codice completo dell'esempio fatto:#

```
// B06orchestra.java (F.Spagna) Esempio di polimorfismo delle classi

public class B06orchestra {

   class musicista {
      void suona() { }
   }
   class violino extends musicista {
      void suona() {
         System.out.println("suona il violino");
      }
   }
   class violoncello extends musicista {
      void suona() {
         System.out.println("suona il violoncello");
      }
```

```
   }
   class pianoforte extends musicista {
      void suona() {
         System.out.println("suona il pianoforte");
      }
   }
   musicista Oistrack = new violino();
   musicista Rostropovich = new violoncello();
   musicista Oborine = new pianoforte();
   Oistrack.suona();            // suona il violino
   Rostropovich.suona ();       // suona il violoncello
   Oborine.suona();             // suona il pianoforte
   musicista trio[] = new musicista[3];

   trio[0] = new violino();
   trio[1] = new violoncello();
   trio[2] = new pianoforte();
   for (int n = 0; n < 3; n++)
      trio[n].suona();
   }
}
```

2.3 *Casting* tra oggetti

Abbiamo visto al paragrafo 2.2.1 che, dove in un programma ci si aspetta una referenza oggetto di una certa classe si può mettere una referenza di un oggetto di una sottoclasse di quella e che non è possibile fare l'inverso, cioè usare una referenza di una superclasse: questo è solo possibile ricorrendo ad un'operazione di casting.

Su un oggetto di una certa classe in Java è infatti possibile fare un'operazione di **casting** mediante la quale un oggetto è convertito in un altro di classe diversa, essendo però il casting permesso soltanto se tra le due classi esiste una relazione di eredità.

Ad esempio, riprendendo l'esempio della classe `Scritta` del paragrafo 2.1.2 e quello della sua classe derivata `scrittaColorata` del paragrafo 2.1.11, di cui si conoscesse un oggetto `miaScrittaRossa`, si può dichiarare una nuova variabile oggetto `miaNuovaScritta` della classe `Scritta` così:

```
scrittaColorata miaNuovaScritta = (scrittaColorata)miaScritta;
```

Con il casting viene a crearsi (senza fare ricorso al `new`) un nuovo oggetto in una nuova classe contenente tutte le informazioni che l'oggetto di partenza possedeva e quello originario continua ad esistere.

Esempio avanti e indietro:#si ritrova?

Riprendendo l'esempio fatto al paragrafo 2.1.11:

```
animale aniFido = (animale)Fido;

cane mioCane = (cane)aniFido;
```

Il casting esplicito non è necessario da un oggetto di una superclasse ad un altro di una sottoclasse, in quanto una sottoclasse ha già in sé tutte le informazioni proprie della superclasse avendole ereditate da essa.

Il casting esplicito diventa invece obbligatorio quando dove è richiesto un oggetto di una certa classe si adopera invece un oggetto di una sua superclasse (casting all'insù) perché comporta una perdita di informazione.

Vediamo l'esempio seguente con le spiegazioni indicate come commenti inseriti nel codice stesso:

```
// B07prova1.java (F.Spagna) Esempi di casting tra classi
                                                        // superclasse
class sup {
   void nome() { System.out.println("A"); }    // metodo della superclasse
}
                                                        // sottoclasse
class sot extends sup {
   void nome() { System.out.println("B"); }    // metodo della sottoclasse
}

public class prova1 {

   public static void main(String[] s) {
      sup a = new sup();                  // crea un'istanza della superclasse
      sot b = new sot();                  // crea un'istanza della sottoclasse
      a.nome();                  // a usa il metodo della superclasse (il suo)
      b.nome();              // b usa il metodo della sottoclasse (ridefinito)

      a = b;                  // casting non necessario e assegnamento lecito
      a.nome();              // a usa il metodo della sottoclasse (quello di b)
      b = (sot)a;//casting obbligatorio e assegn.possibile solo se posto a = b
      b.nome();                           // b usa il metodo della sottoclasse
   }
}
```

Può essere fatto anche il casting di un oggetto verso un'interfaccia se la classe dell'oggeto implementa (direttamente o per eredità da una superclasse) quell'interfaccia. Questo tipo di casting permette di usare i metodi dell'interfaccia [non ho capito#].

2.4 Metodo `equals()` della classe `Object`

La classe `Object` possiede il metodo `equals(Object obj)` che viene ereditato da ogni oggetto in Java. Tale metodo ha come sola istruzione:

```
return this == obj;
```

È interessante notare che il metodo `equals()` riceve come argomento una variabile di tipo `Object`, e quindi può ricevere anche oggetti di qualunque altro tipo in quanto ogni oggetto in Java deriva da `Object`, ma su questo parametro bisogna fare un'operazione di *casting* quale quella riportata nel codice dell'esempio del paragrafo 2.3.1 al momento della verifica dell'identità (anche questa anticipazione risulterà più chiara in seguito).#

2.5 Operatori sugli oggetti e eguaglianza di oggetti

I soli operatori applicabili agli oggetti, e più precisamente alle loro referenze, sono il segno di uguaglianza `==` e quello di diversità `!=`, oltre naturalmente a quello di assegnazione. L'uguaglianza è soddisfatta solo se le due referenze tra cui è posto il segno == puntano allo stesso oggetto.

Nel caso di due stringhe di valore uguale, ad esempio con `s1` e `s2` create con l'operatore `new` nelle due istruzioni:

```
String s1 = new String("abcd");
String s2 = new String("abcd");
```

l'espressione di confronto `s1 == s2` restituisce `false` perché i due oggetti, pur avendo valore uguale, sono due oggetti distinti (ne sono stati creati due).

Per il confronto del valore di due stringhe la classe `String` ha a disposizione il metodo `equals()` (ereditato dalla superclasse `Object`). Ecco perché l'espressione:

```
s1 == s2
```

è concettualmente diversa da quella:

```
s1.equals(s2)
```

che nel caso dell'esempio precedente restituirebbe `true`.

Si è presentato il caso di stringhe, ma le considerazioni fatte sono valide in generale per gli oggetti di ogni classe. Per le stringhe si noti invece in particolare che, se si fossero costruite le due stringhe `s1` ed `s2` con le istruzioni:

```
String s1 = "abcd";
String s2 = "abcd";
```

anzichè con il `new` come si era fatto sopra, `s1` ed `s2` punterebbero effettivamente ad uno stesso oggetto di tipo `String`, ma questo è solo dovuto al fatto che in questo caso particolare il compilatore fa questa scelta per ragioni di ottimizzazione.

A parte i casi sopracitati, il Java non ammette il cosiddetto *overloading degli operatori*, cioè la possibilità di dare significati particolari agli operatori quando essi sono applicati agli oggetti a seconda della loro classe. E ciò per ragioni di semplicità in quanto tale proprietà (pur rimpianta da qualcuno, e presente invece nel C++), soprattutto quando se ne abusa, tende a rendere il codice poco leggibile.

2.6 Determinazione della classe di un oggetto

Per determinare a quale classe appartiene un determinato oggetto si può usare il metodo **`getClass()`** che ogni oggetto eredita dalla classe `Object` e che restituisce un oggetto di tipo `Class`. Siccome poi la classe `Class` ha il metodo `getName()` che restituisce il nome della classe come stringa, si può ottenere questo nome sotto forma di stringa con:

```
String nomeClasse = referenzaOggetto.getClass().getName();
```

Un test sulla classe cui appartiene un oggetto può anche essere fatto con l'operatore **instanceof** che si usa in un'espressione della forma:

```
referenzaOggetto instanceof unaClasse
```

la quale restituisce `true` se `referenzaOggetto` punta ad un'istanza di `unaClasse`, altrimenti `false`.

Quest'operatore può essere usato anche con un'`interface` come operando di destra, per verificare se un oggetto implementa una determinata interfaccia.

2.7 Classi interne (*inner class*)

Nella versione 1.1 del JDK è stata introdotta la possibilità di definire una classe all'interno di un'altra (la classe è allora detta classe interna o ***inner class***), essendo questa soluzione naturale quando nella definizione di una classe si ha bisogno di una classe ausiliaria con un compito limitato: tipico è ad esempio il caso delle classi *listner* nel modello di gestione degli eventi per delega (vedi paragrafo 7.x.x). Per le classi interne si possono distinguere i seguenti casi:

- **classi interne membro** (*member class*) con una definizione del tipo:

```
class classPrinc {

   class classInn {
      // definizione della classe interna
   }
   // altri membri della classe principale (variabili d'istanza)
   // altri membri della classe principale (metodi)
}
```

Per creare un'istanza della classe interna all'interno della classe principale basta adoperare il new:

```
inn i = new inn();
```

ma dall'esterno della classe principale si deve usare un'espressione del tipo:

```
inn i = new miaC.inn();vedi#
```

- **classi interne locali** (*local class*) definite all'interno di un metodo o di un blocco di codice delimitato da due parentesi `{ }`.

- **classi interne anonime** (*anonymous class*) create con un `new` nella stessa espressione in cui esse sono definite senza l'introduzione di un loro nome specifico: in tal caso non è necessaria la presenza di un costruttore. Esempio:#

-

Quando una classe (ad es. `classPrinc`) contenente una classe interna (ad es. `classInn`) è compilata viene creato un file (`classPrinc.class`) relativo alla classe principale ed un file (`classPrinc$1$classInn.class`) relativo alla classe interna.

Un esempio di classe interna adoperata per una classe *listner* di un evento è quello riportato al paragrafo 7.x.x relativo al modello di gestione eventi per delega.

2.8 Copia di oggetti

Per copiare un oggetto in un altro in Java esistono i due metodi `copy()` e `clone()`.

Un esempio di applicazione del metodo **`copy()`** può essere il seguente:

```
Point p1 = new Point();
Point p2 = new Point();
p2.copy(p1);      // copia il valore delle variabili d'istanza di p1 in p2
```

nel quale l'oggetto `p1` è copiato in `p2`, che è un oggetto della stessa classe diverso, ma con gli stessi valori attuali delle variabili d'istanza (quindi con `p2.equals(p1)` di valore `true`).(controlla#)

E' importante notare che se l'oggetto da copiare ha come variabili d'istanza degli altri oggetti, di questi nel nuovo oggetto vengono copiate solo le referenze e non gli oggetti stessi.

Il metodo **`clone()`** viene usato in modo diverso:

```
p2 = (Point)p1.clone();
```

che richiede il casting in quanto con questo metodo l'oggetto `p1` è clonato in un'istanza della classe `Object` ottenuta come valore di ritorno del metodo.

2.9 Interfacce

Un'**interfaccia**, dichiarata con la parola **`interface`**, consiste in un insieme di metodi `public` che si possono applicare tutti insieme ad una classe per attribuirle certe funzionalità, metodi che nella dichiarazione dell'interfaccia stessa sono però dichiarati solo a livello di prototipi con la loro signature e non ancora effettivamente definiti, perchè è previsto che essi debbano essere implementati di volta in volta nella classe cui sono applicati quando essa è definita, in relazione alle esigenze particolari di questa. Le interfacce si dimostrano molto utili per superare a volte la limitazione del Java riguardo all'eredità di una classe che è permessa da una sola altra classe. Quando viene derivata una nuova classe da una superclasse si può anche al momento stesso dell'estensione implementare in essa un'interfaccia, intendendo con ciò farle assumere, insieme a tutte le proprietà ereditate dalla superclasse, anche in più un determinato insieme di nuovi comportamenti.

Le interfacce possono essere dichiarate **`public`** o **`private`** come le classi (`private` è la caratteristica di default, valida quando non si precisa nulla), ed i loro metodi sono sempre `public` e definiti come `abstract` nella dichiarazione d'interfaccia, mentre non saranno evidentemente più tali al momento della loro implementazione nella classe (vedi l'esempio seguente). Le interfacce possono, come le classi, contenere anche delle variabili, ma queste devono essere necessariamente dichiarate `final` in quanto si tratta di costanti. C'è anche chi ha assimilato le interfacce a classi con tutti metodi astratti (`abstract`) e variabili (attributi) `final`, anche se dal punto di vista dell'eredità le due cose sono diverse.

Per implementare (attribuire) un'interfaccia in una classe si specifica nella dichiarazione di classe dopo il nome della classe stessa quello dell'interfaccia facendolo precedere dalla parola chiave **`implements`** e se ne definiscono quindi tutti i metodi. Ecco due esempi:

Definizione di un'interfaccia:

```
public interface miaInterf {
   static final int x = 1;      // una variabile (costante) dell'interfaccia
   public abstract void met();              // metodo sotto forma di prototipo
}
```

Implementazione dell'interfaccia in una classe:

```
public class miaClasse implements miaInterf {
   void met() {                                  // metodo da implementare
      // istruzioni chè implementano il metodo
   }
}
```

Caso con implementazione di un'interfaccia su una classe derivata per eredità:

```
public class miaApplicaz extends java.applet.Applet
                         implements miaInterf {
   void met() {                                  // metodo da implementare
      // istruzioni chè implementano il metodo
   }
}
```

Una classe può implementare contemporaneamente un numero qualunque di interfacce.

Delle nuove interfacce possono essere create per **estensione di altre interfacce**, nel modo qui indicato:

```
public interface miaInterf extends interf1, interf2 {
   // corpo dell'interfaccia
}
```

[# spiega meglio]

Le interfacce possono essere anche usate come tipo nella dichiarazione di una variabile oggetto: se una variabile è dichiarata di un tipo corrispondente a un'`interface` essa può referenziare un oggetto che implementi quell'interfaccia. Esempio#:

```
public class miaClasse {
   // da rivedere
}
```

Oltre a quelle definite nei package di sistema di Java, delle nuove interfacce possono essere utilmente definite ed adoperate in un programma quando si vuole che classi diverse abbiano una comune particolare funzionalità. Una volta che la funzionalità che si vuole generalizzare è stata ben definita, si individuano, nelle loro caratteristiche esterne (cioè nella signature), tutti i metodi di cui essa abbisogna inglobandoli nella definizione di un'interfaccia, ed essi potranno poi essere definiti in modo specifico secondo le circostanze nelle classi che, implementando quell'interfaccia, li dovranno utilizzare.

2.10 Codice sorgente e package

Il **codice sorgente** di un'applicazione Java viene scritto in uno o più file di testo portanti l'estensione `.java`. In generale il codice sorgente Java contenente in uno o più file, sotto forma di testo, le dichiarazioni di un certo numero di classi e di interfacce aventi una relazione di contesto, raggruppate sotto un unico nome, costituisce un **package**. Al package può essere dato

un nome scrivendolo, nella prima istruzione del (o dei) file, di seguito alla parola chiave **package**, ad esempio:

```
package miopack;
```

ma tale istruzione non è obbligatoria: in genere essa viene messa quando si sta creando un insieme di classi (e interfacce) che si prevede di riutilizzare successivamente importandole in altri programmi; se invece l'istruzione è assente, resta comunque il fatto che il file rappresenta sempre un package (in generale ogni classe in Java fa parte di un package), ma non c'è modo di importare in altri package i suoi elementi (classi e interfacce). Un package può essere distribuito anche su più file, cioè due o più file diversi possono riportare in testa lo stesso nome di package (si può andare a vedere come esempio il codice sorgente del JDK, nel quale ogni classe definita occupa un suo file contenente come prima istruzione la denominazione del package). La collocazione del codice in più package aventi un loro nome proprio permette di risolvere i conflitti di nomi di classi, variabili e metodi contenuti in diversi gruppi di classi. Le classi predefinite del linguaggio Java sono contenute in diversi **package di sistema** (input/output, rete, interfaccia grafica).

Ogni file sorgente Java può contenere più di una classe ma di esse una sola può essere dichiarata `public` (e quindi essere usata da classi di altri package) e il file deve avere lo stesso nome di questa (con gli stessi caratteri minuscoli e maiuscoli), però con l'estensione `.java`.

I nomi dei package sono costituiti da uno o più termini separati da punti, il primo termine è di solito quello che si riferisce al creatore del package stesso: nel caso dei package di sistema del JDK di Java il primo nome è `java` e questi package, e solo questi, devono essere obbligatoriamente sempre presenti in ogni implementazione di Java sviluppata da chiunque.

Riportiamo qui un esempio di una classe di sistema di Java:#

Per cominciare a fissare le idee, scriviamo subito in un file chiamato `scriveSeno.java` il codice sorgente Java relativo ad un'applet molto semplice.

```
// B08scriveSeno.java (F.Spagna) Esempio di package

package nostro.nuovoPack;

import java.applet.Applet;          // classe Applet del package java.applet
import java.lang.Math;                  // classe Math del package java.lang

public class scriveSeno extends Applet {
   System.out.println(Math.sin(0.5));
}
```

2.11 Importazione di classi da altri package

Un programma contenente le definizioni di una o più classi e interfacce può importare anche classi appartenenti ad un altro package mediante delle istruzioni di `import`, che, messe all'inizio del file di codice, subito dopo l'eventuale istruzione di `package` e prima della definizione delle classi, sono simili alle direttive di compilazione di tipo `#include` del C e del C++ nel senso che importano del codice (delle classi) definite altrove. Per precisare le classi

importate si fa precedere, unendolo con un punto, il loro nome da quello del package cui esse appartengono.

Riprendendo l'esempio di codice del paragrafo precedente, si possono notare due istruzioni di tipo **`import`**, con le quali vengono incluse nel codice presente classi o interfacce disponibili in altri package.

Le classi del package `java.lang` sono sempre automaticamente accessibili da ogni programma Java e non richiedono quindi che si importi esplicitamente il package nel codice di un programma con un'istruzione di `import`.

Nel caso dell'esempio si sono in particolare importate le due classi `Applet` del package `java.applet` e `Math` del package `java.lang` ambedue della libreria di sistema Java, ma ugualmente avrebbero potuto essere importate, se ce ne fosse stato bisogno, delle classi da un altro package predisposto a parte dal programmatore stesso o da package forniti da terze parti.

Se, anzichè importare una classe specifica di un package, si scrive un asterisco al posto del nome della classe dopo il nome del package, per esempio:

```
import java.applet.*;
```

vengono importate tutte le classi pubbliche del package che sono effettivamente utilizzate nel codice che segue (nel caso dell'esempio il package `java.applet`).

Ma c'è un modo alternativo del tutto equivalente (la scelta di quale usare sta al programmatore secondo le circostanze o le sue preferenze) per dichiarare l'importazione di classi di un altro package, secondo il quale si può far precedere direttamente il nome delle classi da importare dal nome del package cui esse appartengono direttamente nel codice nel punto stesso in cui esse sono richiamate, senza così dover più scrivere l'istruzione di `import` relativa. In questo modo si evitano i conflitti di omonimia (classi, variabili o metodi del package presente o di un package importato aventi lo stesso nome di analoghi elementi dello stesso codice o di altri package importati). Il codice del paragrafo precedente si scriverebbe allora:

```
public class scriveSeno extends java.applet.Applet {
   System.out.print(java.lang.Math.sin(0.5));
}
```

Dal punto di vista della compilazione e dell'esecuzione di un programma l'importazione di un package comporta una prima operazione di tipo `include` fatta in fase di compilazione, per la definizione delle classi, ed un'altra di caricamento e **link dinamico** delle classi fatta al run-time, cioè al momento dell'esecuzione.

Un package importato può a sua volta contenere l'importazione di altre classi e così via.

3. La sintassi del linguaggio

3.1 Elementi del linguaggio (*token*)

I costituenti elementari (*token*) del codice Java possono appartenere a una delle seguenti categorie:

parole chiave, che sono parole speciali riservate dal linguaggio con significato particolare

identificatori, nomi dati a classi, oggetti, variabili, metodi, etc.

letterali, valori numerici, caratteri o stringhe scritti direttamente come tali nel codice

separatori, simboli di separazione tra parti di codice

operatori, caratteri o combinazioni di caratteri che definiscono le diverse operazioni

spazi, a volte necessari per separare certi tokens, a volte solo utili per un codice più chiaro

commenti, scritti dal programmatore sul codice per spiegazioni ma ignorati dal compilatore.

Consideriamo ad esempio l'istruzione seguente che contiene un esempio di ognuno degli elementi suddetti:

```
int i = 5;    // commento
```

`int` è una parola chiave (è quella che definisce gli interi), a destra di tale parola c'è uno spazio (necessario), `i` è un identificatore (nome di una variabile), il segno `=` è un operatore (di assegnamento), `5` è un letterale (esprime il valore numerico 5), il `;` è un separatore (separa un'istruzione dalla successiva) e alla fine c'è un commento (che il programmatore ha voluto aggiungere per sua comodità).

3.2 Istruzioni e blocchi di istruzioni

Un'**espressione** è un elemento di codice costituito da uno o più token (esclusi i separatori e i commenti) che nel suo insieme restituisce un valore. Un'espressione può essere adoperata per assegnare un valore ad una variabile, per fare delle operazioni nell'ambito di un'espressione più grande che la contiene, per passare un valore di un argomento ad un metodo, o per farne dei test se restituisce un valore booleano. Ecco alcuni esempi di espressioni:

```
5, a, "abc", 5+a, true, 5+7*a>0, met(), a=2.
```

Le istruzioni sono terminate sempre con un punto e virgola (`;`) e sono eseguite nel programma in modo sequenziale.

Ogni **istruzione** (*statement*), che costituisce un'operazione più o meno complessa in un programma, termina sempre con un punto e virgola (`;`).

Un'istruzione può anche essere costituita da un semplice punto e virgola (istruzione vuota o nulla) che può essere posto dove andrebbe un'istruzione. Esempio con un loop infinito:

```
while (true);
```

ed un altro con un loop finito:

```
for (int i = 0; i <= 10000; i++) ;
```

o anche:

```
etichetta: ;
```

Una serie di istruzioni può essere racchiusa da due parentesi `{` e `}` venendo così a costituire un **blocco** che può essere messo in ogni posto dove ci può essere una singola istruzione. All'interno di un blocco esiste una visibilità (*scope*) locale in quanto le variabili che vi vengono definite all'interno restano locali al blocco stesso e sono viste solo dentro di esso cessando di esistere all'uscita dal blocco, cioè quando questo à stato completamente eseguito (vedi anche paragrafo 2.1.8).

Dei blocchi di istruzioni sono di solito usati come corpo di un metodo o per circoscrivere una parte di codice interessata da istruzioni di controllo di flusso come `for`, `while`, `do`, `if` o `else`.

3.3 Parole chiave

Le **parole chiave** di Java, riservate dal linguaggio per compiti particolari, sono elencate nella tabella seguente:

```
abstract        package
boolean         private
break           protected
byte            public
byvalue         rest
case            return
cast            short
catch           static
char            super
class           switch
const           synchronized
continue        this
default         threadsafe
do              throw
double          transient
else            true
extends         try
false           var
final           void
finally         while
float
for
future
generic
goto
if
inner
implements
import
instanceof
int
interface
long
native
new
null
operator
```

Vedremo la spiegazione del loro significato mano a mano che esse si presenteranno durante la presentazione del linguaggio.

3.4 Identificatori

Gli **identificatori**, cioè i nomi che il programmatore dà a classi, interfacce, variabili e metodi, possono essere costituiti da **caratteri Unicode**, che comprendono, oltre ai consueti caratteri ASCII, anche altri milioni di caratteri degli alfabeti internazionali (ma dell'insieme di tutti i possibili valori, che vanno da 0x0000 a 0xFFFF, solo però quelli superiori al numero esadecimale 0x00C0), quindi possono essere usate per esempio anche parole accentate. L'uso di caratteri Unicode è forse però utile più che per gli identificatori, per i caratteri di stringhe da scrivere. Naturalmente sono esclusi dall'uso come caratteri per gli identificatori quelli già riservati dal linguaggio per gli operatori (come `+`, `-`, `*`, `%`, `<`, `>`, `|`, `&`, etc.). Gli identificatori possono iniziare, oltre che con una qualsiasi lettera dell'alfabeto, anche con i caratteri `_` (sottolineatura) o `$`, ma non con una cifra. Gli identificatori sono sensibili al minuscolo e maiuscolo (sono cioè, come si dice, "*case-sensitive*") in quanto le lettere maiuscole sono considerate diverse dalle minuscole, così ad esempio `miaclasse` e `miaClasse` sono due identificatori distinti. Non c'è alcun limite alla lunghezza degli identificatori.

A proposito degli identificatori che un programmatore può usare per classi, oggetti, variabili e metodi, si tenga presente che è buona norma adoperare per essi nomi significativi ai fini di una migliore leggibilità del codice (ma non esageratamente lunghi) che aiutano molto nella rilettura del codice, soprattutto se fatta a distanza di tempo o da parte di altri programmatori.

C'è poi un'abitudine diffusa tra i programmatori Java quando si vogliono usare nomi costituiti da più parole, di unire in un identificatore le diverse parole facendo risaltare le singole parole con iniziali maiuscole, eccetto la prima che viene lasciata minuscola: questa convenzione facilita la lettura perché generalmente i programmatori vi sono già abituati. Un esempio sarebbe: `numeroDiElementi`.

3.5 Tipi di variabili

Ogni variabile (paragrafo 3.7) o letterale (paragrafo 3.6) di tipo elementare (quelli di base, non le referenze di oggetti di una classe) appartiene ad uno dei tipi previsti nel linguaggio, che caratterizzano il modo di memorizzazione del valore. I tipi elementari possono essere in generale interi e a virgola mobile, caratteri e booleani.

3.6 Letterali

3.6.1 Che cosa sono i letterali

I letterali sono valori scritti direttamente come tali nel codice, sia che si tratti di numeri, caratteri, stringhe o `true` e `false` booleani.

3.6.2 Letterali interi

Un letterale intero è in generale automaticamente (per *default*) di tipo `int`, ma è di tipo `long` se ha un valore più grande del massimo intero che il tipo `int` può rappresentare (vedi

paragrafo 3.8.2). Un letterale intero può essere forzato al tipo `long` facendolo seguire dalla lettera **`l`** minuscola o **`L`** maiuscola.

Gli interi possono essere espressi anche come ottali (*octal*), cioè numeri espressi in base 8, facendoli precedere da uno **`0`**, o come esadecimali (*hex*), numeri espressi in base 16, facendoli precedere da un **`0x`** o **`0X`**: così per esempio `010` equivale a 8 e `0x10` a 16.

Riportiamo qui degli esempi di letterali interi:

`123`	intero normale
`123l` o `123L`	intero forzato a `long` (seguito da una **`l`** minuscola o **`L`** maiuscola)
`0123`	intero espresso in octal (preceduto da uno zero)
`0x123` o `0X123`	intero espresso in notazione esadecimale (preceduto da **`0x`** o **`0X`**)

3.6.3 Letterali a virgola mobile

I letterali a virgola mobile espressi in un programma sono automaticamente (per *default*) di tipo `double`. Un letterale a virgola mobile può essere forzato ad essere di tipo `float` facendolo seguire una lettera **`f`** (o **`F`**). I letterali possono essere espressi anche come fattori di potenze di 10 mediante una lettera **`e`** che segue il numero e precede il valore dell'esponente, espresso con o senza segno.

Esempi di letterali a virgola mobile:

`1.23`	se scritto così il compilatore lo assume come `double` per default
`1.23f` o `1.23F`	forzato a `float` (se seguito da una **`f`** minuscola o **`F`** maiuscola)
`1.23d` o `1.23D`	forzato a `double` (doppia precisione) (se seguito da **`d`** o **`D`**)
`1.23e4` o `1.23E4`	forma con esponente di 10: 1.23e4 equivale a 1.23×10^4
`1.23de+12`	forma con esponente di 10 e numero forzato a `double`

3.6.4 Letterali booleani

I due soli letterali booleani possibili sono **`true`** e **`false`**.

3.6.5 Letterali carattere

I letterali carattere sono espressi con il segno del carattere compreso tra due apostrofi, come in:

```
char c = 'a';
```

o anche:

```
char c = 'à';
```

ma pure con il numero Unicode di 4 cifre esadecimali precedute da **`\u`**, per esempio:

```
char c = '\u2122';
```

In Java esistono dei **caratteri speciali** preceduti dal segno **** con funzioni particolari (sono gli stessi caratteri speciali del linguaggio C):

`\`	continuazione di riga usabile all'interno di una stringa che nel codice vada a capo in altra riga
`\n`	nuova riga
`\t`	tabulazione
`\b`	backspace (ritorno del cursore indietro dello spazio di un carattere)
`\r`	ritorno carrello (ritorno all'inizio della riga)
`\f`	*"form feed"*
`\\`	\ (barra rovescia)
`\'`	' (apostrofo)
`\"`	" (virgolette)
`\ddd`	ottale
`\xdd`	esadecimale
`\udddd`	carattere Unicode

Esempio:#

3.6.6 Letterali di tipo `String`

I letterali di tipo `String` sono costituiti da una sequenza di caratteri, racchiusi da virgolette, che possono comprendere anche spazi o caratteri speciali. Ad esempio:

```
"questo e\' un letterale di tipo String"
```

3.7 Variabili

Le variabili sono caratterizzate da un tipo, un nome (il loro identificatore) ed un valore dal momento in cui questo è loro assegnato.

Ad ogni variabile corrisponde uno spazio di memoria della macchina fisso, di grandezza dipendente dal tipo di variabile, con un certo contenuto (valore della variabile) che può variare nel corso del programma.

Ad ogni occorrenza del nome di una variabile il programma durante la sua esecuzione vi sostituisce il suo valore corrente.

Ogni variabile deve essere sempre dichiarata, dopodichè le può essere assegnato un valore che viene immagazzinato nell'area della memoria che il compilatore riserva alla variabile al momento della sua dichiarazione.

Le variabili in Java esistono solo nell'ambito di una classe e possono essere variabili d'istanza, variabili di classe e variabili locali. Delle prime due si è già parlato nel capitolo 2. Le **variabili locali** sono quelle dichiarate dentro i metodi per un uso interno ai metodi stessi e hanno visibilità soltanto al loro interno e non sono visti all'esterno: quando l'esecuzione del

metodo è completata esse cessano di esistere. Se una variabile locale è dichiarata all'interno di un blocco racchiuso dalle parentesi { e } essa è vista solo all'interno del blocco e termina di esistere all'uscita del blocco stesso. Quando più di un metodo nell'ambito di una classe deve trattare con una stessa variabile allora più che variabili locali si usano variabili d'istanza, che sono viste da tutti i metodi contemporaneamente.

La dichiarazione di una variabile va fatta facendo precedere al suo nome il tipo, per esempio:

```
int i;
float f;
char c;
boolean b;
```

La posizione di una dichiarazione di variabile può essere qualsiasi nell'ambito del metodo in cui viene fatta, purchè preceda l'uso della variabile stessa in altre istruzioni. Però nella maggioranza dei casi si è soliti dichiarare le variabili all'inizio del metodo per averle tutte bene sotto controllo.

Diverse variabili dello stesso tipo possono essere anche dichiarate in un'unica istruzione, come per esempio:

```
int i, j, m, n;
```

Le variabili possono essere inizializzate (cioè può essere loro assegnato un valore iniziale) al momento stesso della dichiarazione, per esempio:

```
int i = 2;
int j = 3, m, n = 4;
```

Prima di essere usata, una variabile locale deve avere già necessariamente un valore assegnato con un'inizializzazione fatta già al momento della dichiarazione o con un'assegnazione specifica successiva: se si dimentica di farlo il compilatore stesso ne fa richiesta esplicita. Le variabili d'istanza sono invece inizializzate automaticamente dal compilatore a valori di zero per le variabili di tipo numerico, al carattere `'\0'` per quelle di tipo carattere, a `false` per i booleani e a `null` per le referenze ad oggetti di classi.

3.8 Variabili di base

3.8.1 Variabili di base (o primitive) e loro tipo

Le **variabili di base** o **elementari** o **primitive** non sono veri e propri oggetti, in quanto il loro comportamento si distacca da quello degli oggetti Java. Esse rappresentano una deroga alla regola generale secondo la quale in Java ogni entità è un oggetto, e questo è dovuto a un'esigenza di efficienza. Vedremo come ci sia comunque la possibilità di rappresentare i tipi elementari con oggetti usando delle classi speciali (*object wrapper*). Il formato di ogni tipo di variabile di base (o primitiva) in Java è definito in modo preciso e non può dipendere dalla macchina, come invece succede per altri linguaggi. Quindi mai due macchine diverse possono produrre risultati differenti con lo stesso codice.

Le variabili primitive possono essere di tipo intero, a virgola mobile, booleano o carattere.

Java non presenta un tipo a virgola fissa (*fixed point*).

3.8.2 Variabili intere

Le **variabili intere**, che sono le variabili che possono assumere valori espressi da numeri interi, in Java possono essere di quattro tipi, secondo la larghezza della loro rappresentazione in numero di bit:

`byte`	a 8 bit	(può assumere valori compresi tra -128 e +127)
`short`	a 16 bit	(può assumere valori compresi tra -32 768 e +32 767)
`int`	a 32 bit	(può assumere valori compresi tra -2 147 483 648 (-2^{31}) e +2 147 483 647 ($2^{31}-1$))
`long`	a 64 bit	(può assumere valori compresi tra –9 223 372 036 854 775 808 (-2^{63}) e +9 223 372 036 854 775 807 ($2^{63}-1$))

Questi tipi hanno un segno, cioè possono comprendere valori positivi o negativi.

In Java non esiste un tipo di intero senza segno (l'*unsigned integer* del C).

Il programmatore decide il tipo particolare di intero da usare per una variabile intera in base ai valori che presumibilmente la variabile assumerà nel corso del programma; se poi invece accadrà che il suo valore superi il valore massimo ammissibile per il tipo scelto esso verrà troncato. Per esempio:#

Un troncamento avviene anche quando si fa il cast di un tipo intero di una certa capacità verso un tipo meno capace: ne risulta il modulo rispetto alla lunghezza massima del tipo di arrivo. Se per esempio:

short s = 1000;(#VEDI?)

byte b = (byte)s;

b assume il valore di:

3.8.3 Variabili a virgola mobile

Le variabili **a virgola mobile** (***floating point***), che sono quelle il cui valore può contenere una parte decimale, possono essere, a seconda del modo di rappresentazione, di tipo a singola precisione o a doppia precisione:

`float`	a 32 bit	valori compresi tra 1,4012984643 e^{-45} e 3,4028234663 e^{+38} (positivo o negativo)
`double`	a 64 bit	valori compresi tra 4,9406564584 e^{-324} e 1,7976931348 e^{+308} (positivo o negativo)

I `float` hanno circa 7 cifre significative e i `double` ne hanno circa 15.
I tipi di dati a virgola mobile di Java rispondono alla norma IEEE 754 e quindi trattano i valori **`NaN`** e **`Inf`** (vedi paragrafo 3.9.11).

3.8.4 Variabili booleane

In Java esiste anche un tipo **`boolean`** di 1 bit (che è assente in C e nel primo C++), che può essere o vero o falso, cioè assumere come soli valori possibili quello di **`true`** o di **`false`**, che sono parole chiave che esprimono i due possibili letterali di tipo booleano.

Si noti che dove nel codice il compilatore si aspetta un tipo booleano non si può mettere un'espressione numerica, come si può fare (e si fa comunemente) invece in C ed in C++, perchè Java distingue nettamente e separa i tipi tra di loro, così che ad esempio `0` non ha niente a che vedere con `false`, come `1` non l'ha con `true`.

I booleani non possono essere sottoposti a conversione di tipo (casting).

3.8.5 Variabili carattere

Il tipo di variabile **carattere** in Java ha a disposizione una parte dell'insieme di caratteri **Unicode** (*Unicode Worldwide Character Standard*) a due byte (16 bit), essendo di quelli però utilizzabili soltanto 34168 caratteri distinti:

`char` a 16 bit senza segno.

Notiamo che in Java, come d'altra parte in C, i caratteri sono rappresentati da numeri (a 16 bit in Java, quando in C erano a 8 bit) e quindi sarebbero come degli interi `short`. Ma non possono essere usati come tali (come era invece in C e C++) neanche con un casting verso interi.(#VEDI?)

Non è però detto che se si chiede ad un programma Java di stampare sullo schermo un carattere definito con il suo valore numerico (come indicato al paragrafo 3.6.5), esso sia sempre correttamente stampato, in quanto sta al sistema operativo occuparsi dell'uscita sullo schermo e la stampa sarà possibile solo se il sistema operativo supporta lo standard Unicode.

3.9 Operatori

3.9.1 Tipi di operatori

Gli operatori possono essere distinti in relazione al numero di operandi cui si applicano in ***unari*** o ***binari*** (ma ce n'è anche uno ***ternario***) a seconda che si applicano ad uno, a due (o a tre) operandi. Essi possono essere di assegnamento, aritmetici, di incremento e decremento, e logici.

3.9.2 Operatore di assegnamento

L'operatore di assegnamento `=` serve per assegnare un valore ad una variabile che sia stata già dichiarata e che quindi abbia una memoria riservata ad essa che possa contenerne il valore: prima l'espressione a destra del segno uguale viene calcolata e poi il risultato viene assegnato alla variabile posta a sinistra (chiamata perciò *left value*) del segno `=`. Ad esempio:

```
a = 2;  b = 3;  c = a + b;
```

L'operazione di assegnamento stessa rappresenta nel suo insieme un'espressione che restituisce il valore stesso assegnato in essa, così che, se si scrive ad esempio:

```
b = (a = 3);
```

poiché le operazioni di assegnamento successive procedono a partire da destra verso sinistra, l'espressione `a = 3` restituisce il valore `3`, e questo valore viene quindi assegnato a `b`: in tal modo sia `a` sia `b` sono posti uguale a 3 e anche tutta l'espressione dell'intera istruzione ha questo stesso valore. Abbiamo racchiuso il primo assegnamento (quello di destra) tra parentesi per motivi di chiarezza della spiegazione, ma le cose non cambiano se si scrive invece:

```
b = a = 3;
```

Nel caso di un assegnamento del tipo:

```
a = a + 2;
```

in cui la variabile `a` compare sia a destra sia a sinistra del segno uguale, prima viene valutato il valore a destra del segno uguale con il valore che la variabile `a` ha fino a quel momento e cioè subito prima dell'istruzione in questione, e poi viene assegnato questo valore ad `a` come suo nuovo valore.

3.9.3 Operatori aritmetici

Gli operatori aritmetici, agenti su due operandi (variabili o letterali) tra i quali sono interposti, sono:

+	addizione	esempio:	`a + 1`
-	sottrazione	esempio:	`b - 2`
*	moltiplicazione	esempio:	`5 * a`
/	divisione	esempio:	`8 / 3`
%	modulo	esempio:	`c % 10`

L'operatore meno (`-`) può anche essere usato davanti ad una variabile numerica come suo singolo operando per cambiare segno al suo valore.

L'operazione di divisione tra due interi restituisce un intero ed ignora il resto della divisione: ad esempio `16 / 3` restituisce il valore di 5.

L'operatore modulo (`%`) restituisce invece il resto della divisione dei due operandi cui viene applicato, così che `16 % 5` restituisce il valore di 1.

Il risultato di un'operazione aritmetica tra interi è sempre generalmente un intero, che sarà di tipo `int` o `long` solo in relazione alla grandezza del valore del risultato e indipendentemente dal tipo particolare degli operandi: è per default di tipo `int` qualunque sia il tipo degli operandi, anche se essi sono di tipo `byte` o `short`, ma diventa di tipo `long` se il risultato oltrepassa il massimo valore che può essere rappresentato da un `int` o se almeno uno degli operandi è di tipo `long`.

Se il risultato di un'operazione tra interi oltrepassa il valore massimo del tipo specifico (per esempio un `int` cui viene assegnato un valore maggiore di 65 536) esso viene posto uguale al modulo riferito al valore massimo detto. Si ha quello che viene chiamato *overflow* e wraps (prova#)...

Il risultato di un'operazione tra un operando intero di qualunque tipo e uno a virgola mobile è sempre un valore a virgola mobile (secondo i casi `float` o `double`).

3.9.4 Operatori di incremento e decremento

In Java sono disponibili i simboli di operatori **++** e **--** tipici del C che possono essere usati rispettivamente per incrementare o decrementare di 1 il valore di una variabile dopo la quale siano posti. Quindi:

a++; equivale a: **a = a + 1;**

```
a--;    equivale a:    a = a - 1;
```

Gli operatori di incremento e decremento possono però essere anche posti prima del nome della variabile:

```
++a;    --a;
```

e la differenza dei due modi di applicare gli operatori (dopo o prima del nome della variabile) consiste nel fatto che nel primo caso la variabile è prima usata con il valore che aveva fino a quel momento nelle istruzioni precedenti, per calcolare l'espressione che la contiene e poi viene incrementata, quindi l'incremento avrà effetto solo dopo l'istruzione in cui esso è indicato, mentre nel secondo caso (operatore prefisso) la variabile è prima incrementata e poi subito usata con il nuovo valore nella stessa espressione. A seconda dei casi vi può essere o non essere una differenza tra i due modi: se per esempio si scrivono istruzioni semplici come:

```
a++;    e    ++a;
```

il risultato è lo stesso; ma se invece si scrive:

```
a = 1;    b = a++;
a = 1;    c = ++a;
```

`b` vale 1 ma `c` vale 2.

Ma non sempre le cose sono così semplici perché in caso di espressioni complesse si possono manifestare effetti collaterali per cui l'espressione `a++` non è sempre del tutto equivalente a `a = a+1`.

Un'espressione equivalente, ma abbreviata, di quella di assegnamento del tipo:

```
a = a + b;
```

che Java (come d'altra parte il C) offre è la seguente:

```
a += b;
```

che equivale alla precedente.

Così anche:

```
a -= b;    equivale a:    a = a - b;
a *= b;    equivale a:    a = a * b;
a /= b;    equivale a:    a = a / b;
a %= b;    equivale a:    a = a % b;
```

E così per esempio, se `a = 16` l'espressione `a /= 5` restituisce il valore di 3 e quella `a %= 5` restituisce 1.

Ma l'equivalenza dei due tipi di espressioni, quello completo e quello abbreviato, può venire a cadere in certi casi particolari di espressioni complesse in conseguenza di effetti collaterali.

3.9.5 Precedenza degli operatori

Quando più operatori sono presenti nella stessa espressione questa è valutata dando un ordine di precedenza agli operatori secondo regole ben definite, essendo il risultato in generale

dipendente dall'ordine di applicazione degli operatori. Tipica di tutti i linguaggi di programmazione è per esempio la precedenza di moltiplicazione e divisione su somma e sottrazione che è naturalmente rispettata anche in Java. L'ordine di precedenza è in generale il seguente:

- incremento e decremento
- operatori aritmetici
- operatori di confronto
- operatori logici
- assegnazioni.

L'ordine in cui le operazioni sono fatte può essere cambiato mediante l'uso di parentesi che possono racchiudere le espressioni che si vuole siano valutate prima, con la possibilità anche di annidare parentesi dentro altre, essendo le espressioni racchiuse tra le parentesi più interne valutate prima di quelle più esterne. A volte, anche quando l'uso di parentesi non è necessario per stabilire un determinato ordine di priorità, si possono inserire delle parentesi per migliorare la chiarezza e la leggibilità del codice. Così ad esempio uno potrebbe scrivere:

```
if ((a == 0) && (b != 2))
```

anche se le parentesi più interne non sono necessarie stanti le regole fissate per la precedenza degli operatori.

Qui di seguito si riporta una tabella con l'elenco degli **operatori** posti in ordine di precedenza, partendo da quella più alta e finendo a quella più bassa.

Gli operatori allo stesso livello di precedenza sono adoperati procedendo da sinistra a destra nell'ordine in cui appaiono nell'espressione.

Tabella 3 Ordine di precedenza degli operatori

```
.    [ ]   ( )
++   --    !     ~
*    /     %
+    -
<<   >>    >>>>
<    >     <=    >=
==   !=
&
^
|
&&
||
? :
=    +=    -=    *=    /=    &=    |=    ^=    %=    >>=    <<=    >>>>=
```

In Java, come d'altra parte anche in C, non esiste un operatore di elevazione a potenza: un'operazione di questo tipo può essere fatta con il metodo `pow()` della classe `Math` del package `java.lang`.

3.9.6 Operatori di confronto

Gli operatori di confronto, che servono per paragonare due operandi restituendo un valore booleano (`true` o `false`), sono elencati nella tabella seguente:

`==`	restituisce `true` se il valore del primo operando è uguale a quello del secondo operando (da distinguere dal segno di assegnazione `=`)
`!=`	restituisce `true` se il valore del primo operando è diverso da quello del secondo operando
`>`	restituisce `true` se il valore del primo operando è maggiore di quello del secondo operando
`<`	restituisce `true` se il valore del primo operando è minore di quello del secondo operando
`>=`	restituisce `true` se il valore del primo operando è maggiore di o uguale a quello del secondo operando
`<=`	restituisce `true` se il valore del primo operando è minore di o uguale a quello del secondo operando

Nel caso in cui gli operatori `==` e `!=` siano usati con operandi referenze di oggetti ... (vedi anche paragrafo #).

3.9.7 Operatore condizionale ternario ? :

Un'espressione della forma:

```
espressioneBooleana ? valore1 : valore2
```

che si applica a tre termini (e perciò l'operatore è detto ternario), può essere interpretata come un **if ... then ... else ...** e restituisce il valore `valore1` se l'espressione booleana `espressioneBooleana` è vera (cioè è uguale a `true`), altrimenti restituisce `valore2`. I termini `valore1` e `valore2`, che possono essere dei letterali, ma più in generale espressioni qualsiasi, devono essere dello stesso tipo.

Per esempio l'espressione:

```
max = x > y ? x : y
```

restituisce il valore massimo di `x` e `y`. Oppure:

```
a = (a >= 0)? a : -a;
```

è un'istruzione che assegna ad `a` il suo valore assoluto. Infatti prima viene valutata l'espressione booleana `a >= 0` e se essa è vera l'espressione a destra del segno di assegnazione (`=`) è posta uguale ad `a`, altrimenti è posta uguale a `-a`, e l'uno o l'altro di questi valori sono assegnati alla variabile `a` a sinistra del segno `=`.

Un altro esempio potrebbe essere il seguente: ammettiamo che un metodo `met()` si aspetti un valore come parametro e noi vogliamo dare ad esso al momento della chiamata del metodo un valore dipendente da una certa variabile booleana (chiamiata ad esempio `a`), allora si può fare il test al momento stesso in cui si invoca il metodo e scrivere, in modo molto compatto, qualcosa come:

```
met(x ? a : b);
```

Per esempio per un colore che debba essere rosso per valori positivi di `x` o altrimenti bianco:

```
setColor(x > 0 ? Color.red : Color.white);
```

3.9.8 Operatori logici

Gli operatori logici, che agiscono su operandi di tipo booleano (variabili, letterali o più in generale espressioni che restituiscono valori booleani), e producono risultati essi stessi di tipo booleano, sono qui sotto elencati e possono essere combinati insieme in espressioni più complesse.

Unari:

`!`

operazione di NOT logico: inverte il valore dell'unico operando booleano cui si applica (una variabile, un letterale o più in generale un'espressione) da `false` a `true` e da `true` a `false`: così per esempio `!true` equivale a `false`.

Binari:

`&` o `&&`

operazione di AND logico: l'espressione è vera solo se ambedue gli operandi sono veri, altrimenti è falsa: la differenza tra i due tipi sta nel fatto che con `&&` se l'espressione di sinistra è falsa l'intera espressione restituisce `false` e l'espressione di destra non viene più valutata, con `&` invece le due espressioni vengono comunque sempre valutate entrambe.

`&=`

assegna al primo operando il risultato di un AND logico con il secondo operando.

`|` o `||`

operazione di OR logico: l'espressione è vera se almeno uno dei due operandi è vero: la differenza tra i due tipi sta nel fatto che con `||` se l'espressione di sinistra è vera l'intera espressione restituisce `true` e l'espressione di destra non viene più valutata, con `|` invece le due espressioni vengono comunque sempre valutate entrambe. Generalmente per le espressioni logiche vengono usati gli operatori `&&` e `||` piuttosto che `&` e `|`, che invece sono usati con operandi interi come operatori logici sui bit.

`|=`

assegna al primo operando il risultato di un OR logico con il secondo operando

`^`

operazione di XOR logico o OR esclusivo: l'espressione è vera solo se i due operandi sono diversi, quindi uno dei due vero e l'altro falso.

`^=`

assegna al primo operando il risultato di uno XOR logico con il secondo operando.

Ecco un esempio con operazioni sui booleani:

```
boolean b;   b = (a > 0);   if (b == !true) b = !b;
```

L'ultima istruzione (un po' improbabile a dire il vero e inutilmente complicata, ma riportata per avere una prima idea di possibili operazioni sui booleani) va a vedere che valore ha la variabile booleana b e se la trova falsa la fa diventare vera.

3.9.9 Operatori sui bit

Gli operatori sui bit (*bitwise operators*), tutti binari, cioè agenti su due operandi, sono adoperati per fare operazioni sui singoli bit di numeri interi. Eccone l'elenco:

&	operazione di AND sui bit di due interi tra cui il segno è posto
|	operazione di OR sui bit di due interi tra cui il segno è posto
^	operazione di XOR (OR esclusivo) sui bit di due interi tra cui il segno è posto
<<	spostamento a sinistra nel primo operando di un certo numero di bit
>>	spostamento a destra nel primo operando di un certo numero di bit
>>>	spostamento a destra nel primo operando di un certo numero di bit con riempimento di zeri
~	operazione di NOT binario (complemento)
<<=	assegnazione ad una variabile del suo valore dopo spostamento a sinistra di n bit
>>=	assegnazione ad una variabile del suo valore dopo spostamento a destra di n bit
>>>=	assegnazione ad una variabile del suo valore dopo spostamento a sinistra di bit
&=	assegnazione ad una variabile del suo valore dopo un'operazione di AND
|=	assegnazione ad una variabile del suo valore dopo un'operazione di NOT
^=	assegnazione ad una variabile del suo valore dopo un'operazione di XOR

Si fa qui un esempio in cui viene analizzata la composizione binaria di un numero intero utilizzando gli operatori sui bit:#

3.9.10 Tabella riassuntiva degli operatori su variabili intere

Operatori unari:

`-` cambia segno all'operando che lo segue

`~` fa il complemento bit per bit dell'operando che lo segue

`++` incrementa di 1 l'operando (prima del suo uso se lo precede, dopo l'uso se lo segue)

`--` sottrae il valore 1 all'operando (prima dell'uso se lo precede, dopo se lo segue)

Operatori binari:

`+` fa l'addizione dei due operandi tra cui è posto

`+=` assegna al primo operando la somma del primo con il secondo operando

`-` fa la sottrazione dei due operandi tra cui è posto

`-=` assegna al primo operando la differenza tra il primo e il secondo operando

`*` fa la moltiplicazione dei due operandi tra cui è posto

`*=` assegna al primo operando il prodotto del primo per il secondo operando

`/` fa la divisione dei due operandi tra cui è posto

`/=` assegna al primo operando il risultato della divisione del primo per il secondo operando

`%` fa il modulo del primo operando rispetto al secondo operando

`%=` assegna al primo operando il valore del modulo del primo rispetto al secondo operando

`&` operazione di AND sui bit di due interi tra cui il segno è posto

`&=` assegnazione ad una variabile del suo valore dopo un'operazione di AND

`|` operazione di OR sui bit di due interi tra cui il segno è posto

`|=` assegnazione ad una variabile del suo valore dopo un'operazione di NOT

`^` operazione di XOR (OR esclusivo) sui bit di due interi tra cui il segno è posto

`^=` assegnazione ad una variabile del suo valore dopo un'operazione di XOR

`~` operazione di NOT binario (complemento)

`<<` fa lo scorrimento a sinistra dei bit del primo operando di un numero di posti uguale al secondo operando

`>>` fa lo scorrimento a destra dei bit del primo operando di un numero di posti uguale al secondo operando

`<<=` assegna alla variabile rappresentata dal primo operando il risultato dello scorrimento a sinistra dei bit del primo operando di un numero di posti uguale al secondo operando

`>>=` assegna alla variabile rappresentata dal primo operando il risultato dello scorrimento a destra dei bit del primo operando di un numero di posti uguale al secondo operando

`>>>` spostamento a destra nel primo operando di un certo numero di bit con riempimento

	di zeri
`>>>=`	assegnazione ad una variabile del suo valore dopo spostamento a sinistra di bit
`>>>>`	fa lo scorrimento a destra dei bit del primo operando di un numero di posti uguale al secondo operando con riempimento di zeri a sinistra
`>>>>=`	assegna al primo operando il risultato dello scorrimento a destra dei bit del primo operando di un numero di posti uguale al secondo operando con riempimento di zeri a sinistra

3.9.11 Operazioni su variabili a virgola mobile

Alle variabili a virgola mobile possono essere applicati gli stessi operatori visti per gli interi, con le seguenti particolarità:

- non sono applicabili gli operatori che agiscono sui bit;
- gli operatori `++` e `--` aggiungono o sottraggono all'operando il valore 1.0 anzichè 1 intero;
- l'operatore modulo (`%`) dà come risultato il resto della divisione in virgola mobile.

Un'espressione a virgola mobile contenente variabili di tipo `float` dà un risultato di tipo `float`, ma se almeno una delle variabili contenute nell'espressione è di tipo `double` il risultato sarà di tipo `double` perchè prevale la precisione maggiore (vedi anche il paragrafo 3.9.11).

La divisione per zero di una variabile a virgola mobile dà come risultato **Inf** e non è segnalato un errore, come avverrebbe invece nel caso di una variabile intera. Un altro valore che può assumere una variabile a virgola mobile è **NaN** (**N**ot **a** **N**umber). Tutto questo è conforme alla specifica IEEE 754.

3.9.12 Operazioni tra variabili di tipo diverso

Se in un'espressione sono fatte delle operazioni tra variabili di tipo diverso, per esempio tra interi e variabili a virgola mobile, il compilatore, prima di fare il calcolo, trasforma i tipi più deboli nel tipo più forte contenuto nell'espressione e tutta l'espressione viene valutata in questo tipo, dove l'ordine dei tipi, partendo dal più debole e arrivando al più forte, è il seguente: `byte, short, int, long, float, double`. Questo permette per esempio di scrivere degli interi (scritti ad esempio: `2`) in un'espressione contenente delle variabili a virgola mobile, ma in questo caso è comunque preferibile scrivere i numeri, anche se per loro natura sono interi, con il punto come se fossero a virgola mobile (quindi scrivendoli come: `2.`) per ragioni di chiarezza di lettura. Ma questo diventa obbligatorio se si hanno degli interi a denominatore in una divisione (vedi#).

Quanto detto sopra riguarda il calcolo che viene fatto di un'espressione, ma non ha niente a che fare con l'assegnazione del valore calcolato ad una variabile posta a sinistra del segno `=` (variabile detta proprio perciò **lvalue** (*leftvalue*, cioè valore di sinistra), che conserva sempre il tipo fissato una volta per tutte nella sua dichiarazione qualunque cosa avvenga a destra dell'espressione. Se il risultato dell'espressione a destra (ad esempio `int`) è di tipo più debole del tipo dell'lvalue (che può essere ad esempio `float`), prima dell'assegnazione esso viene automaticamente trasformato al tipo del lvalue (da `int` a `float`), e se invece il risultato a destra è di tipo più forte dell'lvalue il compilatore segnala un errore perché questo comporta una

perdita di informazione, ma tale errore può essere evitato con un'operazione di casting (vedi paragrafo 3.14).

3.9.13 Operatori su stringhe

Alle stringhe, che in Java sono veri e propri oggetti appartenenti alla classe **String** , possono essere applicati gli operatori + e += che producono il concatenamento di due stringhe (questo, per chi conosce il C++, si può dire che sia l'unico caso in Java di *overloading di un operatore* nel senso che viene dato al segno + un significato aggiuntivo a quello che esso ha già per la somma di due valori numerici). Ad esempio l'istruzione:

```
String s = "Mario" + " Rossi";
```

crea una nuova stringa `s` di valore `"Mario Rossi"`.

Se un operando di un'operazione di questo tipo è un numero, esso viene prima convertito nella stringa che rappresenta a caratteri il numero, ad esempio con:

```
String s = "Siamo nel " + 1998;
```

la stringa `s` vale `"Siamo nel 1998"`.

Ma non si ottiene l'inizializzazione della stringa scrivendo semplicemente:

```
String s = 1996;
```

senza che il numero sia aggiunto ad un'altra stringa. (vedi ""+...#)

Ma c'è di più, in quanto è lecito usare il segno + tra una stringa ed un oggetto qualsiasi, perché in tal caso viene creata una nuova stringa che è formata con il concatenamento della prima stringa con la stringa restituita dal metodo `toString()` dell'oggetto. Ogni oggetto Java ha infatti un metodo di questo tipo ereditato dalla classe `Object`, che, come si sa, è la capostipite di ogni classe Java, e tale metodo in generale nella sua forma originaria restituisce il nome della classe seguito da brackets (?#), ma può restituire un'altra stringa se il metodo è stato implementato per overriding (in altre parole è stato ridefinito) nella classe in questione per restituire una determinata stringa.

L'operatore **+=** riproduce l'analogia del += tra operandi di tipo numerico, e infatti `s += "ab"` è un'espressione che restituisce `s + "ab"` e quindi:

```
s += "ab";
```

equivale a:

```
s = s + "ab";
```

3.9.14 Operatori su oggetti

Gli oggetti hanno a disposizione l'operatore **instanceof** che verifica, restituendo un valore booleano (`true` o `false`), se un oggetto è o no un'istanza di una determinata classe. In generale esso viene usato in istruzioni del tipo:

```
if (mioOgg instanceof unaCertaClasse) ...
```

3.10 Commenti

Si noti nel codice del paragrafo 2.1.2 anche il modo di inserire i **commenti** (cioè quelle parti scritte sul codice per commentarlo, ma ignorate dal compilatore) facendoli precedere da due barre `//`, uguale a quello del C++: il compilatore considera come commento e quindi ignora tutto quello che segue questo segno fino alla fine della riga.

In Java è disponibile comunque anche il modo, proprio del C, di racchiudere un commento tra i segni `/*` (che apre il commento all'inizio) e `*/` (che chiude il commento alla fine) e questo modo è usato soprattutto per commenti che comprendono varie righe. Un commento di questo tipo non può contenere a sua volta annidato al suo interno un altro commento di questo stesso tipo, ma puo' contenere uno o più commenti del tipo `//`.

Esempi:

```
a = 1;    // questo e' un commento


/*
    e anche questo e' un commento
*/
```

Ma il Java presenta anche un terzo tipo di delimitazione di un commento, che può essere racchiuso tra i simboli `/**` (all'inizio) e `*/` (alla fine), e può essere usato per la generazione automatica di documentazione (mediante l'applicazione `javadoc` fornita con il JDK stesso), nella quale il commento posto prima di una dichiarazione comparirà come testo.

3.11 Stringhe

Le ***stringhe***, consistenti in una serie di caratteri, che possono comprendere anche degli spazi e caratteri speciali (vedi paragrafo 3.6.5), sono in Java degli oggetti della classe **`String`**, che dispone di alcune variabili d'istanza rappresentanti le loro caratteristiche e di vari metodi per il loro trattamento (ai programmatori C e C++ può essere fatto notare che le stringhe in Java non sono array di caratteri come in quei linguaggi).

Dal punto di vista della loro creazione, le stringhe hanno un comportamento particolare in quanto classe, potendo essere istanziate, oltre che, come ogni altro oggetto, mediante l'operatore `new`, anche semplicemente assegnado loro un letterale stringa, come in:

```
String s = "una stringa";
```

Se in un certo punto del codice si scrive un'espressione con un certo numero di caratteri compresi tra doppie virgolette, ad esempio:

```
"questo e\' un letterale stringa\n"
```

che costituisce un letterale di tipo `String`, si viene contestualmente a istanziare un oggetto della classe `String` il cui valore è inizializzato con la successione di caratteri dati. E questo giustifica espressioni del tipo:

```
if ("miaStringa".equals(altraStringa)) ...
```

in quanto già con lo scrivere `"miaStringa"` si crea un oggetto della classe `String` che, come ogni altro oggetto in Java, dispone del metodo `equals(...)`, che può quindi essere invocato come è stato fatto.

E' da osservare che una stringa vuota, espressa da:

```
String s = "";
```

non è la stessa cosa di una stringa di valore `null`, cioè di una stringa cui non è stato assegnato ancora alcun valore (stringa non inizializzata) o cui sia stato assegnato esplicitamente il valore `null`.

Per i metodi e le operazioni sulle stringhe si veda il paragrafo 4.13.5.

3.12 Array

In Java gli **array** (vettori), che rappresentano una serie ordinata di variabili, che ne costituiscono gli **elementi**, tutte dello stesso tipo (variabili elementari o referenze di oggetti) e caratterizzate ognuna da un nome comune e da un indice progressivo che parte da 0, sono (contrariamente a C e C++) dei veri e propri oggetti appartenenti ad una classe, anche se questa classe non ha un suo nome specifico (la classe `Array` esiste, ma è un'altra cosa). Il numero di elementi di un array rappresenta la sua **dimensione**.

Gli array sono dichiarati facendo seguire al nome del tipo di variabile le due parentesi `[]` vuote, come negli esempi seguenti:

```
int[] i;    float[] f;    String[] s;    Punto[] p;
```
(per la nostra classe `Punto`)

che significano che `i`, `f`, `s` e `p` sono oggetti (o istanze) di classe tipo array rispettivamente di tipo `int`, `float`, `String` e `Point`, anche se il nome della classe non viene scritto esplicitamente e sono le due parentesi quadre che caratterizzano la classe come array. Ma le istruzioni precedenti possono anche essere scritte in modo diverso, ma del tutto equivalente, aggiungendo le parentesi `[]` non al nome del tipo, ma al nome della variabile:

```
int i[];    float f[];    String s[];    Punto p[];
```
(per la nostra classe `Punto`)

ma questa notazione, pur molto usata, è forse meno conforme della precedente al modo con cui si fa la dichiarazione per altri tipi di oggetti.

La semplice dichiarazione però, come avviene per ogni altro oggetto, non crea ancora l'array, dice soltanto di che tipo esso è. Un oggetto array è creato mediante l'operatore **`new`**, con un'istruzione della forma seguente:

```
i = new int[10];  f = new float[10];  s = new String[10];
p = new Punto[10];
```

nella quale viene data una dimensione definita all'array, permettendo così al compilatore di allocare la memoria relativa ai suoi elementi.

La dichiarazione, la creazione e il dimensionamento di un array possono essere fatti anche contemporaneamente con un'unica istruzione della forma:

```
int[] i = new int[10];  String[] s = new String[10];
```

oppure, alternativamente:

```
int i[] = new int[10];  String s[] = new String[10];
```

La dimensione dell'array nelle espressioni precedenti però non deve necessariamente essere una costante, come quella degli esempi fatti, ma può essere rappresentata da una variabile o un'espressione di tipo intero il cui valore è determinato durante l'esecuzione del programma.

La classe rappresentante gli array deriva direttamente dalla classe `Object`, tuttavia da essa non possono essere derivate altre classi in quanto la classe è `final`. Essa, in quanto classe, può avere delle variabili d'istanza, e infatti ha in particolare una sua variabile `public` **`length`**, che assume il valore del numero di elementi dell'array (pertanto l'espressione `i.length` degli array degli esempi precedenti vale 10).

Ogni **elemento** di un array è caratterizzato dal nome dell'array e da un indice scritto tra parentesi `[` e `]` che rappresenta la posizione dell'elemento `i[n]` nell'array. Gli elementi del vettore del primo esempio sono quindi rappresentati individualmente da:

```
i[0], i[1], i[2], . . . i[9]
```

Siccome gli indici partono da zero, come nel C e nel C++, l'ultimo (cioè il decimo) elemento dell'array di dimensione 10 definito sopra, è l'elemento `i[9]`. L'indice di un elemento può essere rappresentato da una qualsiasi espressione che restituisca un intero.

Se un'espressione restituisce un array, un elemento, ad esempio quello di indice `n`, dell'array può essere specificato come `espressione[n]`, così che per un metodo che restituisce un array si può avere un'espressione della forma:

v.espressione in libro (o metodo qui) #

```
metRestitArray()[n];
```

Mentre un array di tipo elementare contiene effettivamente le variabili costituenti i suoi elementi, un array di oggetti in Java è in realtà un array i cui elementi memorizzano referenze agli oggetti (qualcosa come gli array di puntatori del C/C++) e non gli oggetti stessi.

Si noti bene che l'istanziazione di un array di oggetti crea l'array in quanto array di referenze, ma tali referenze esistono ma sono inizialmente di valore `null`, cioè non puntano ancora ad alcun oggetto. Perché esse puntino ad oggetti veri si devono creare questi oggetti ed assegnare le loro referenze agli elementi dell'array. Esempio:#

Si osservi l'istanziazione dell'array di referenze prima e quella degli oggetti puntati da esse poi.

Un **array a due dimensioni** (matrice) in Java può essere rappresentato da un array di array, con una dichiarazione del tipo:

```
int i[][] = new int[10][5];      o alternativamente:
int[][] i = new int[10][5];
```

o anche soltanto:

```
int i[][] = new int[10][];      o alternativamente:
int[][] i = new int[10][];
```

dove la seconda dimensione può essere definita e la memoria allocata in un secondo tempo. Analogamente per array a più dimensioni.

L'**inizializzazione** di un array può essere fatta successivamente alla sua creazione con operazioni di assegnazione di ciascuno dei suoi elementi, oppure anche al momento stesso della dichiarazione, nel seguente modo, che crea l'array senza richiedere l'uso esplicito dell'operatore `new`, dimensiona implicitamente l'array al numero di elementi espressi tra le parentesi `{` e `}` e inizializza i suoi elementi con i valori indicati. Per esempio:

```
int i[] = { 1, 5, 7, 10 };
```

equivale a:

```
int i[];
i = new int[4];
i[0] = 1; i[1] = 5; i[2] = 7; i[3] = 10;
```

Così anche un array a due dimensioni può essere dichiarato e allo stesso tempo inizializzato con un'istruzione del tipo:

```
int i[][] = {
    { 1, 2, 3, 4 },      // primo indice = 0, secondo indice da 0 a 3
    { 2, 4, 6, 8 },      // primo indice = 1, secondo indice da 0 a 3
    { 3, 6, 9, 12},      // primo indice = 2, secondo indice da 0 a 3
};
```

(si noti l'ordine con cui progrediscono il primo ed il secondo indice: per fissarlo nella memoria si potrebbe pensare il primo indice come quello di ciascuna riga e il secondo come quello della successione su ogni riga dei dati scorsi in sequenza).

Gli elementi degli array sono al momento della creazione dell'array inizializzati automaticamente quelli numerici a valori di zero, i caratteri a valori di `'\0'`, le variabili booleane a valori di `false` e le referenze ad oggetti a valori `null`.

Gli array sono sottoposti in fase di compilazione da parte del compilatore ed in fase di esecuzione (*run-time*) da parte dell'interprete a un controllo degli indici, così che nel corso del programma l'indice di ogni elemento dell'array trattato non deve mai superare i limiti definiti nella dichiarazione dell'array: ogni tentativo di uscire da essi produce un errore (o meglio un'eccezione di tipo `ArrayIndexOutOfBoundsException` viene sollevata: vedi paragrafo 5.4). Sotto quest'aspetto il Java è decisamente più sicuro dei linguaggi C e C++, che invece non effettuano alcun controllo sugli array, con rischi di uscita inavvertita dallo spazio di memoria assegnato e conseguenze imprevedibili e spesso disastrose. Per gli array a più di una dimensione è segnalato un errore se almeno in una delle sue dimensioni viene superato il limite fissato nel dimensionamento dell'array per la dimensione stessa. Il controllo al run-time è efficace quando l'indice di un elemento è calcolato nel corso dell'esecuzione del programma in relazione ad esempio a calcoli o a input dell'utente. Per evitare il rischio di oltrepassare i limiti dell'array da parte di un indice si ha a disposizione la variabile d'istanza `length` dell'array sulla quale nel programma possono essere fatti dei test.

Per copiare un array in un altro (anche soltanto parzialmente) si può adoperare il metodo **`System.arraycopy()`**.

3.13 Inizializzazione di un array di oggetti

L'inizializzazion e di un array di stringhe può essere fatto nella forma:

```
String s[] = { "lunedi", "martedi", "mercoledi" };
```

che è analoga a quella della creazione di una stringa mediante la sua inizializzazione con un letterale stringa.

Un array di oggetti di una certa classe può essere inizializzato con referenze ad oggetti della stessa classe dichiarata dell'array, ma anche ad oggetti di classi derivate da quella classe.

Per esempio è possibile scrivere:

```
Object a[][] =  { { "rosso", Color.red   } ,
                  { "verde", Color.green } ,
                  { "blu",   Color.blue  } };
```

in quanto sia gli oggetti `String` sia quelli `Color` con cui si fa l'inizializzazione sono derivati dalla classe `Object`, e quindi le loro referenze possono essere poste laddove è prevista una referenza ad oggetti di tipo `Object` (vedi paragrafo 2.3 sul casting degli oggetti). Questo permette di usare oggetti di classi diverse nello stesso array, il che può essere a volte utile.

Nell'esempio precedente gli elementi dell'array sono di tipo `Object` e quindi se poi si vogliono avere gli oggetti dell'array come `String` e `Color` sarà necessario fare un casting di classe sugli elementi dell'array (vedi paragrafo 2.3):

```
(String) a[n][0]

(Color) a[n][1]
```

3.14 Conversione di tipo (*casting*) di variabili elementari

In un'espressione contenente variabili di tipo diverso, salvo certi casi, è obbligatorio fare la conversione di tipo esplicita tra le variabili verso un tipo unico. La conversione di tipo (o ***casting***) è un'operazione mediante la quale il valore di una variabile, o più in generale di un'espressione, di un certo tipo viene convertito in un tipo diverso, cioè un tipo elementare viene convertito in un altro tipo elementare oppure un oggetto in un oggetto di classe diversa. Questa conversione si ottiene facendo precedere il nome della variabile che si vuole convertire dal nome del tipo che si vuole ottenere racchiuso da parentesi.

La forma dell'operazione di casting è:

```
(nuovoTipo)espressioneVecchioTipo;
```

Ad esempio la terza delle tre istruzioni:

```
int n;
float f;
f = (float)n;
```

trasforma con `f` l'intero `n` in un `float`.

Questa conversione è sottoposta a certe regole e limitazioni. Come si era visto al paragrafo 3.9.11, per quanto riguarda la conversione dei tipi elementari la conversione di tipo verso valori di tipo più grande, cioè senza perdita di informazione sul valore (da `byte` a `char`, `short` o `int`, da `int` a `long`, da `int` a `float`) il casting non è necessario. In caso inverso il casting esplicito è obbligatorio in quanto esso comporta una perdita di informazione.

Il casting non è necessario tra variabili di tipo intero (che siano di tipo `byte`, `short`, `int` o `long`) e tra `float` e `double`, valendo in questi casi le regole enunciate al paragrafo 3.9.11.

prove nei due sensi

Operazioni di casting sono possibili solo:

- tra variabili numeriche (intere e a virgola mobile);
- tra oggetti legati da una relazione di eredità, cioè tra classi e loro sottoclassi (vedi esempio al paragrafo 2.3.3).

Nessun altro casting è ammesso, per esempio non è lecito fare il casting tra variabili di tipo elementare ed oggetti o viceversa e in ciò il Java è più sicuro di C e C++, che invece permettono qualunque tipo di casting.

3.15 Dichiarazioni e visibilità di variabili ed oggetti

Le dichiarazioni di variabili ed oggetti possono essere poste, all'interno di una classe, in una qualunque posizione del codice, anche se di solito si preferisce metterle all'inizio per averle tutte facilmente sotto controllo. Variabili ed oggetti conservano una visibilità (cioè la proprietà di essere riconosciuti) locale limitata all'interno del **blocco di codice** (porzione di codice compresa tra le due parentesi { e } , entro il quale essi siano stati dichiarati (variabili locali al solo blocco, ove il blocco può essere anche tutto il corpo del metodo in cui sono definiti). Inoltre, se in un'istruzione `for` si scrive per esempio:

```
for (int n = 0; n <= nmax; n++)
   istruzione; ( o blocco di istruzioni entro { } )
```

la variabile `n` esiste ed è visibile soltanto all'interno del blocco del `for`, finito il quale non esiste più.

3.16 Istruzioni di controllo di flusso del programma

3.16.1 if e if ... else

Con l'istruzione **`if (...)`** se una certa condizione (quella posta entro parentesi dopo l' `if`) è verificata, cioè se l'espressione logica booleana che essa esprime è uguale a `true`, il programma esegue l'istruzione o il blocco di istruzioni racchiuso da parentesi { } che seguono, altrimenti li salta e passa alle istruzioni successive. La sintassi è:

```
if (espressioneBooleana)
   istruzione singola;
```

e nel caso di più istruzioni da eseguire se la condizione è verificata:

```
if (espressioneBooleana) {
   blocco di più istruzioni
}
```

L'`if` può essere seguito da un **`else`** se si definisce un'istruzione o un blocco di istruzioni da eseguire come alternativa a quella dell'`if` nel caso che la prima condizione non sia verificata:

```
else
   istruzione; ( o blocco di istruzioni entro { } )
```

Esempio:

```
if (a == true)                               // che si può anche scrivere if (a)
   istruzione1; ( o blocco di istruzioni entro {} )  // eseguita se a è true
else
   istruzione2; ( o blocco di istruzioni entro {} ) // eseguita se a è false
```

A chi proviene dal C o C++ si fa osservare che in Java l'espressione da verificare (quella entro parentesi) deve essere di tipo booleano e non può essere di tipo intero come era invece permesso in quei linguaggi, che sono più permissivi in fatto di tipizzazione.

Esiste una forma alternativa all'`if` usata per espressioni semplici, rappresentata dall'operatore condizionale ternario ? : (vedi paragrafo 3.9.8).

3.16.2 switch

Con l'istruzione **`switch`** `(espressioneIntera)` un'espressione (quella posta entro parentesi dopo la parola chiave `switch`) viene calcolata ed il suo valore confrontato successivamente con una serie di valori che possono essere costanti o forniti da altre espressioni, per scegliere l'operazione da fare, consistente in una o più istruzioni indicate in corrispondenza della condizione che è verificata. Un'importante limitazione è costituita dal fatto che sia il valore su cui fare il test sia il valore di confronto possono essere solo di tipo `byte`, `short`, `int` o `char` e non possono essere né `long`, né `float`, né oggetti. La serie di condizioni (clausole), ciascuna caratterizzata dalla parola chiave **`case`** che precede il valore da verificare (questo seguito da due punti (:)), è racchiusa tra due parentesi { e } . La sintassi è:

```
switch (espressione) {
   case espressione1:
      una o più istruzioni;      // eseguite se espressione == espressione1
      break;
   case espressione2:
      una o più istruzioni;      // eseguite se espressione == espressione2
      break;
   . . .
   default:                                             // facoltativo
      una o più istruzioni;                 // eseguite in ogni altro caso
}
```

dove l'ultima clausola, che è comunque facoltativa, è introdotta dalla parola **`default`** e indica le operazioni da fare nel caso che nessuna delle clausole che la precedono sia verificata.

Esempio:

```
switch (a - b) {
   case 1:
      istruzioni1;
      break;
   case 2:
      istruzioni2;
      break;
```

```
   . . .
   default:
      istruzioni3;
      break;
}
```

Ogni clausola generalmente termina con un **break** che fa uscire dall'intera istruzione di switch dopo l'esecuzione delle operazioni corrispondenti ad essa; se però il break è omesso viene eseguita anche la clausola successiva, indipendentemente dal valore del case in essa indicato e così via finchè non si incontra un break. Questo può essere utilizzato se si vuole ad esempio che una stessa operazione sia eseguita per due o più condizioni diverse. Per esempio:

```
switch (espressione) {

   case espressione1:                                    // primo caso
      istruzioni1;              // eventuali istruzioni limitate al 1° caso
                                                   // qui manca il break
   case espressione2:                                  // secondo caso
      istruzioni2;                         // istruzioni comuni ai due casi
      break;
   . . .
   default:
      istruzioni3;
      break;
}
```

In tal caso le istruzioni2 sono eseguite sia per il verificarsi dell'espressione1 sia per il verificarsi dell'espressione2.

Esempio:#

Nel caso in cui devono essere fatti una serie di confronti con valori di tipo diverso da quelli ammessi (cioè diversi da byte, short, int o char), non potendosi usare un'istruzione di tipo switch, si dovrà ricorrere ad un if annidato del tipo:

```
if (espressione1)
   istruzioni1;
else if (espressione2)
   istruzioni2;
else if (espressione2)
   istruzioni3;
else if (espressione2)
   istruzioni4;
else
   istruzioni5;
```

3.16.3 Ciclo for

L'istruzione **for (...)** ripete ciclicamente (in inglese si parla di *loop*) un'istruzione, o un blocco di istruzioni racchiuso dalle parentesi { } finchè una certa condizione è verificata. In essa vengono indicate tra parentesi ( ) dopo la parola for tre espressioni separate da punti e virgola:

- la prima è un'istruzione da eseguire all'inizio dei cicli, che costituisce in generale, ma non necessariamente, un'inizializzazione, la maggior parte delle volte di un indice,

- la seconda è una condizione booleana di iterazione che all'inizio di ogni ciclo verifica se i cicli devono essere ancora proseguiti (se l'espressione è `true`) o se si è alla fine dei cicli (quando l'espressione è diventata `false` in seguito all'evoluzione dei valori delle variabili che vi figurano),

- e la terza è un'istruzione eseguita ad ogni ciclo (ad esempio tipicamente l'incremento di un indice).

La sintassi è:

```
for (istr.iniz.; condiz.booleana fine ciclo; istruz.ripet.ogni ciclo)
   istruzione; ( o blocco di istruzioni entro {  } )
```

Nell'esempio seguente:

```
int[] = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = i;
```

l'istruzione viene ripetuta in ciclo a partire da `i = 0` fino a `i = 9`, incrementando l'indice di 1 ogni volta. L'indice `i` è dichiarato nell'istruzione stessa di inizializzazione del ciclo e conserva visibilità solo all'interno del loop (come si era già detto al paragrafo 3.15), cioè cessa di esistere alla fine di esso.

Una, due o anche tutte e tre le espressioni entro le parentesi che seguono il `for` possono anche mancare. Ad esempio l'istruzione:

```
for (;;)
   { /* una o più istruzioni */ }
```

produce un loop senza fine. Infatti se manca la seconda espressione è assunto che la condizione di continuazione delle iterazioni sia sempre vera (`true`) ed in queste condizioni il loop può terminare solo per una condizione posta all'interno del blocco con l'azionamento di un `break`.

Sia l'espressione di inizio ciclo (la prima) sia quella dell'operazione da fare ad ogni ciclo (la terza) possono essere costituite da espressioni multiple aventi come separatore una virgola:

Per esempio:

```
for (int i = 0, j = 10; i < 10; i++, j--)
   a[i] = b[j];
```

si noti che in tal caso sia `i` sia `j` hanno vita solo entro il loop.

Un loop `for` può contenere un altro loop (e così via per più loop annidati uno dentro l'altro), il loop interno essendo eseguito ad ogni ciclo del loop esterno. Ecco un esempio (costruzione di una matrice tipo tavola pitagorica):

```
int a[][] = new int[10][10];
for (int i = 1; i <= 10; i++)
   for (int j = 1; j <= 10; j++)
      a[i][j] = i * j;
```

Si può anche usare un'istruzione vuota (costituita quindi dal solo punto e virgola) dove il `for` richiede l'istruzione o il blocco di istruzioni da eseguire, in tal caso tutto si svolge nelle espressioni tra parentesi che seguono il `for`. Pur se questa forma è raramente usata, se ne può tuttavia fare un esempio:

```
for (int i = 0; i < 10; a[i++] = 0);
```

3.16.4 Ciclo `while`

Con l'istruzione **`while`** `(espressioneBooleana)` si ripete ciclicamente un'istruzione, o un blocco di istruzioni racchiuso dalle parentesi `{ }`, che segue, fintantochè una condizione logica posta tra parentesi dopo la parola chiave `while`, ***verificata all'inizio di ogni ciclo***, sia soddisfatta (cioè se l'espressione vale `true`). Quando l'espressione booleana di controllo, in seguito all'evoluzione dei valori delle variabili che vi figurano, diventa `false` il ciclo non viene più eseguito e si esce dal loop. La sintassi è:

```
while (condizione booleana di fine ciclo)
   istruzione; ( o blocco di istruzioni entro { } )
```

Esempio:

```
int n = 0;
while (n++ < 10)
   a[n] = n;
```

Si noti che in un loop `while` la condizione di ciclaggio potrebbe essere trovata `false` già subito al primo ciclo: in tal caso le istruzioni del `while` non sono mai eseguite, e in ciò il `while` differisce dal `do ... while` (vedi paragrafo seguente), che fa sempre almeno la prima iterazione.

Dove si può usare un loop di tipo `for` può essere usato anche un loop di tipo `while`. Per esempio:

```
for (int i = 0; i < 10; i++)
   // istruzioni
```

equivale a:

```
int i = 0;
while (i++ < 10)
   // istruzioni
```

L'uso dell'una o dell'altra forma è una scelta del programmatore, secondo il caso o le sue preferenze.

3.16.5 Ciclo `do ... while`

Con l'istruzione **`do ... while`** `(espressioneBooleana)` si ripete ciclicamente un'istruzione, o un blocco di istruzioni racchiuso dalle parentesi `{ }`, che segue, fintantochè una condizione logica posta tra parentesi dopo la parola chiave `while` sia soddisfatta (cioè se l'espressione vale `true`), e in ciò l'istruzione è simile al `while` del paragrafo precedente, ma con la differenza che ***la condizione è verificata*** non all'inizio ma ***alla fine di ogni ciclo***, e pertanto, diversamente dal `while` semplice, almeno un ciclo viene sempre effettuato. La sintassi è:

```
do
   istruzione; ( o blocco di istruzioni entro { } )
while (condizione booleana di fine ciclo)
```

Ed eccone un esempio (avendosi una sola istruzione da eseguire ad ogni ciclo, le parentesi { e } avrebbero potuto essere omesse):

```
int i[] = new int[10];
int n = 0
do {
   a[n] = n;
} while (n++ <= 10)
```

3.16.6 Label (etichetta)

Un **label** (etichetta), rappresentata da un identificatore seguito da due punti (:), serve per localizzare un punto del codice cui poter fare riferimento per eventuali rinvii del flusso del programma da un `break` o da un `continue`. La sintassi è:

```
nomeLabel: istruzioni;
```

Degli esempi possono essere visti al paragrafo 3.16.7.

Qualche volta le etichette vengono usate per numerare le righe di codice, come nell'esempio che segue:

```
1:   /*
2:     esempio di righe numerate con etichette
3:   */
4:
5:   class miaClas {
6:      int n;
7:      void met() { /*...*/ }
8:   }
```

nel quale esse non hanno alcuna funzionalità programmatica anche se sono sintatticamente corrette, ma questo uso non è molto diffuso e comunque limitato a funzioni didattiche.vedi#

In Java non c'è un'istruzione di tipo `goto`, il cui uso è sempre stato sconsigliato, e ormai quasi completamente abbandonato, anche nei linguaggi che ne presentano una (come il Fortran dove per diverso tempo è stato diffusamente usato), perché porta ad un codice aggrovigliato, non strutturato e di difficile lettura e manutenzione. Tuttavia la parola **`goto`** è una parola riservata del linguaggio (non si sa mai...).

3.16.7 `return`, `break` e `continue`

Il **`return`** è un'istruzione che fa uscire da un metodo e si occupa eventualmente di assegnare il valore che il metodo restituisce all'esterno quando è stato eseguito. Essa può seguire la sintassi:

```
return espressione;
```

oppure semplicemente:

```
return;
```

Nel primo caso il `return` conclude il metodo entro cui esso è posto, assegnando come valore che il metodo stesso restituisce all'esterno quando viene chiamato il risultato dell'espressione, che deve evidentemente essere dello stesso tipo di ritorno del metodo, e nel

secondo caso, che viene utilizzato solo in metodi di tipo `void`, chiude il metodo, che non restituisce nulla. In ogni caso, alla conclusione del metodo il controllo del programma, cioè il proseguimento dell'esecuzione, ritorna all'istruzione che segue quella della chiamata del metodo.

Un flusso ciclico ripetuto (*loop*) tipo `for`, `while` o `do...while` può essere interrotto, oltre che dalla condizione di fine cicli, anche per mezzo di istruzioni tipo `break`, `continue` o `return` inserite nel corpo del loop. L'istruzione:

```
break;
```

chiude completamente il loop in cui è inserita o un `case` di uno `switch` e passa alle istruzioni che seguono il loop (o lo `switch`), mentre invece l'istruzione:

```
continue;
```

adoperata sempre in un flusso ciclico, interrompe soltanto il ciclo in corso, riprendendo con il ciclo seguente.

Nel caso in cui il `break` o il `continue` si trovino in un loop annidato in un altro loop, al `break` si ritorna al ciclo in corso del loop esterno (o quello esterno più vicino se i loop annidati sono più di due).

Al `break` e al `continue` si può aggiungere il riferimento ad un'etichetta (vedi paragrafo 3.16.6), per farvi il rinvio del flusso del programma con istruzioni della forma:

```
break etichetta;
continue etichetta;
```

Questa opzione può essere utile ad esempio se si vuole che un `break` porti fuori non solo dal ciclo in cui è presente, ma anche da un eventuale altro ciclo più esterno.

Esempio:#

4. Programmi Java

4.1 Applicazioni ed applet

Come si è già detto nel capitolo 1, i programmi Java possono essere delle applicazioni o delle applet. Vediamo adesso come essi vengono trattati dal punto di vista della programmazione.

Le **applicazioni Java** sono programmi funzionanti indipendentemente da un browser (programmi *stand-alone*), che sono lanciati da linea di comando ed eseguiti dal sistema operativo esattamente come qualunque altra applicazione scritta in qualsiasi altro linguaggio, anche se richiedono un interprete per l'esecuzione.

Le **applet** sono invece programmi che possono essere ospitati ed eseguiti su una pagina HTML resa visibile da un navigatore (browser), o da un programma apposito chiamato **appletviewer**, e richiamati sulla pagina tramite un *tag* particolare. Esse sono trasmesse (*downloaded*) attraverso la rete World Wide Web (Internet) da un server HTTP per essere eseguite sulla macchina locale (client) del visitatore Web tramite un interprete Java integrato nel browser, se questo ne ha uno, ma possono anche essere solo presenti sul disco del sistema locale sul quale sono viste. Come le applicazioni, anche esse possono interagire con le operazioni (di tastiera e mouse) fatte dall'utente sul sistema locale (client). Un'applet può anche funzionare come componente inserito in una finestra di un'applicazione Java indipendente.

Le applet, potendo provenire esternamente da qualunque posto, sono soggette, per ragioni di sicurezza del sistema locale, a certe restrizioni contro possibili danni (come virus, violazioni della sicurezza o della riservatezza delle informazioni) cui le semplici applicazioni Java non devono sottostare. Tali restrizioni, poste in modo più o meno severo dai diversi browser, impongono in generale alle applet di:

- non poter leggere o scrivere su file del sistema locale,

- non poter eseguire programmi sul sistema locale,

- non poter caricare programmi nativi della piattaforma del sistema locale (come delle DLL).

- non poter aprire connessioni con altri server al di fuori di quello stesso che ha inviato l'applet,

Queste restrizioni, che limitano in modo serio la funzionalità delle applet, riducono però drasticamente i pericoli di danno al sistema client per quanto possibile, ma non si può avere comunque la certezza assoluta che esse non possano essere aggirate da un abile programma che sia capace di aprire una breccia alla sicurezza. D'altra parte già la concezione dello stesso linguaggio Java rende difficile ogni tentativo di attentare alla sicurezza della macchina su cui le applet e le applicazioni sono eseguite con severi controlli sia a livello di compilatore sia a livello dell'interprete.

Essendo le applet ospitate da un ambiente grafico (quello del browser) per scrivere dei caratteri sullo schermo devono fare ricorso a funzioni grafiche, mentre nelle applicazioni basta usare semplici funzioni di output sullo schermo di tipo `System.print()` in formato testo.

Un programma Java può però anche essere allo stesso tempo un'applicazione ed un'applet se viene scritta rispettando le caratteristiche necessarie per l'una e per l'altra funzionalità: esso si comporterà allora diversamente a seconda dei casi adattandosi al contesto in cui agisce.

4.2 Ambiente di sviluppo di Java (JDK)

4.2.1 Il JDK 1.2

L'ambiente di sviluppo di Java chiamato **Java Development Kit** (**JDK**) nella versione 1.2 può essere scaricato dal sito Web di Sun all'URL *http://java.sun.com/products/jdk/1.2/* sotto la forma di un file che per Windows 95/98/NT ha il nome `jdk12-win32.exe` ed è di 20.521.166 byte. Tale file eseguibile, mandato in esecuzione, va ad installare il software che si presenta con il sistema di directory mostrato nella figura 4.1, che occupa in tutto 43,5 MB.
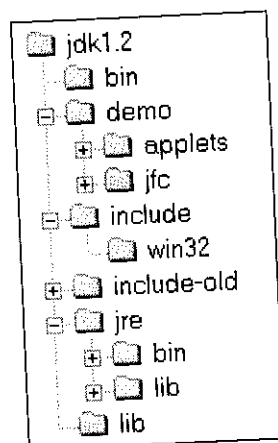
Figura 4.1 Directory del JDK 1.2.

La documentazione è disponibile sullo stesso sito Web e consta del file zippato `jdk12-doc.zip` di 16.897.834 byte.

4.2.2 Il CLASSPATH

Perchè il Java possa trovare le classi di cui ha bisogno per la compilazione o per l'esecuzione deve sapere dove andarle a trovare e cioè in quale directory esse siano situate e ottiene questa informazione da una particolare variabile d'ambiente del sistema operativo chiamata CLASSPATH. Questa variabile viene posta uguale ad una serie di directory, separate tra di loro da un punto e virgola in DOS e in due punti in Unix, con un comando come quello dell'esempio seguente:

```
SET CLASSPATH=.\;C:\jdk1.2\lib\classes.zip;D:\miadir\classi
```

per il DOS (comando generalmente messo nell'`autoexec.bat`), oppure:

```
set CLASSPATH=./:/usr/jdk_base/lib/classes.zip;/u/spagna/miadir/classi
```

per Unix (comando generalmente messo nel file `.profile` personale).

Si noti in particolare che nel path degli esempi è stato compreso anche un file compattato `.zip`: infatti sia il compilatore `javac`, sia il sistema di run-time `java` sono in grado di trovare le classi anche all'interno di file con compressione di tipo zip. E i file di sistema di Java sono archiviati tutti, per ragioni di spazio sul disco, proprio in un unico file di tipo zip, chiamato `classes.zip`.

Ma è anche possibile fornire al compilatore Java queste informazioni non in modo permanente come è fatto nei casi precedenti, ma solo temporaneamente al momento della compilazione mediante l'opzione `-classpath` di complazione:

```
javac -classpath .\;C:\jdk1.2\lib\classes.zip;D:\miadir\classi file.java
```

nel caso del DOS, e:

```
javac -classpath ./:/usr/jdk_base/lib/classes.zip;/u/spagna/dr/classi a.java
```

nel caso di Unix.

4.3 Codice sorgente e compilazione

Il codice sorgente, scritto in linguaggio Java in uno o più file di testo, per un'**applet** deve contenere una classe `public` derivata da `java.applet.Applet`; se invece si tratta di un'**applicazione** la classe principale può essere qualsiasi ma deve avere necessariamente un metodo chiamato `main()`. In un'applet il metodo `main()` non viene scritto esplicitamente perchè risiede nel sistema di run-time del browser. Però se si vuole creare un'applet che possa funzionare anche come applicazione a sè stante basta dotarla anche di un metodo `main()`.

Il nome del file contenente il codice sorgente deve essere uguale a quello della classe (o dell'unica classe `public` se ce n'è più di una nello stesso file) in esso definita, cui è aggiunta l'estensione `.java`.

La compilazione di un codice sorgente Java, che si tratti di un'applet o di un applicazione, viene fatta comunque usando un **compilatore Java**, che per il **Java Developer's Kit (JDK)** di Sun (vedi paragrafo 9.x#) si chiama **`javac`** e viene azionato con il comando seguente:

```
javac miaAppl.java
```

Se non si manifestano errori di compilazione vengono allora prodotti uno o più file binari, ***uno distinto per ogni classe*** presente nel codice sorgente, ciascuno avente un'estensione `.class`, che nel loro insieme compongono l'applet o l'applicazione voluta sotto forma di bytecode (nel caso dell'esempio sarà la `miaAppl.class`). In caso di presenza di errori questi vengono indicati dal compilatore.

L'**opzione di compilazione** **`-O`**, cioè il comando:

```
javac -O miaAppl.java
```

produce un codice più veloce ma di dimensioni maggiori (è un'eventuale scelta da fare in base alle esigenze particolari).

4.4 Creazione ed esecuzione di un'applicazione Java

4.4.1 Creazione di un'applicazione Java indipendente in modalità testo

Un'**applicazione Java** indipendente, contrariamente ad un'applet, non ha bisogno di essere inserita in un documento HTML e di un browser per essere eseguita, ma si appoggia

direttamente al sistema operativo, pur richiedendo comunque il sistema di run-time Java con cui interagisce direttamente.

Un'applicazione Java è costituita da una o più classi, tra le quali quella principale di partenza che deve essere `public` e deve presentare necessariamente un metodo chiamato **`main()`** (che in inglese vuol dire *principale*) così definito:

```
public static void main(String args[]) {
   // corpo del metodo main()
}
```

che è il punto di partenza dell'esecuzione dell'applicazione, in quanto viene chiamato per primo automaticamente dall'interprete al momento dell'esecuzione dell'applicazione, che riceve come argomento è un array di stringhe (elementi di tipo `String`), che rappresenta i parametri che l'utente può passare dalla riga di comando all'applicazione al momento dell'esecuzione. Esso è sempre `public`, `static` e `void`: deve essere `public` per poter essere chiamato da ogni posto, `static` perchè è un metodo di classe e come tale non richiede necessariamente l'istanziazione di un oggetto per essere adoperato ma è richiamabile soltanto come appartenente alla classe (vedi paragrafo 2.1.9), e `void` perché non restituisce nulla. Tale metodo è simile alla funzione `main()` del C, ma è inserito in una classe, perchè, essendo Java un linguaggio compiutamente OOP, tutto vi si svolge nell'ambito di classi).

Un primo esempio molto semplice di applicazione Java potrebbe essere il seguente:

```
// D01applicCiao.java (F.Spagna) Primo esempio di un'applicazione Java

public class D01applicCiao {
   public static void main(String args[]) {
      System.out.println("Ciao");
   }
}
```

consistente nella definizione di una classe, la classe `applicCiao`, contenente un metodo `main()` dove l'unica istruzione è quella che scrive la stringa `"Ciao"` sull'output standard (che è in generale lo schermo, ma potrebbe essere anche qualcos'altro a seconda del contesto in cui l'applicazione agisce, per esempio ...#) utilizzando il metodo `println(...)` dell'oggetto `out` membro della classe `System`, che scrive appunto sull'output standard del sistema. Si noti che è possibile chiamare direttamente il metodo dalla classe `System` senza averne dichiarato alcun oggetto essendo `println(...)` un metodo statico della classe.

Il codice si può limitare ad una dichiarazione di classe senza un'istanziazione esplicita di un oggetto, perché è l'interprete Java che se ne occupa) e così sarà per le applet di cui si occupa, come vedremo, il browser.

Poiché il `main()` è `static` e quindi un metodo di classe, il programma può essere eseguito senza che la classe sia istanziata, ma in questo modo, poichè il metodo `main()` è statico, in esso non si può fare riferimento a variabili d'istanza non statiche o invocare metodi non statici della classe: se si vuole invece trattare l'applicazione come un oggetto avente le sue variabili d'istanza ed i suoi metodi non statici bisogna istanziare un oggetto della classe stessa entro il metodo `main()`, perciò di solito si crea un'istanza della classe stessa nel `main()` (v.libro Lemay pag.28). Esempio:

```
// D02applicCiao.java (F.Spagna) Applicazione istanziata nella classe stessa

public class D02applicCiao {
   String s;                                             // variabile d'istanza
   applicCiao2() { s = "Ciao"; }  // metodo che usa una variabile d'istanza
   public static void main(String args[]) {
      D02applicCiao ma = new D02applicCiao();
      System.out.println(s);
   }
}
```

combina i due#

```
// D03miaApp.java  F.Spagna  Esempio di applicazione Java

public class D03miaApp {
   public void metodo() {
      System.out.println("invocato metodo non statico");
   }
   public static void main(String s[]) {
      D03miaApp mA  = new D03miaApp();
      mA.metodo();
   }
}
```

il cui output è:

#

L'applicazione può anche essere prima definita come classe interna nel `main()` della classe principale ospitante e poi istanziata in esso:

```
// D04ex.java (F.Spagna) Applicazione come classe interna definita e
// istanziata nel main() della classe ospite principale

public class D04ex {
   public static void main(String s[]) {
      class c {
         String st = "val";
         c() { System.out.println(st); }
      }
      new c();
   }
}
```

Se invece la classe è definita come classe interna fuori del `main()`, allora deve essere definita come `static`:

```
// D05ex.java (F.Spagna) Applicazione come classe interna definita fuori del
// main()

public class D05ex {
   static class c {
      String st = "val";
      c() { System.out.println(st); }
   }
   public static void main(String s[]) {
      new c();
   }
}
```

Se le variabili dell'applicazione sono definite come variabili di classe si può fare a meno di creare un oggetto della classe nel `main()`:

```
// D06ex.java (F.Spagna) Un'applicazione con membri statici non istanziata

public class D06ex {
   static String st = "valore";
   public static void main(String s[]) {
      System.out.println(st);
   }
}
```

Il **nome dell'applicazione**, cioè del file con estensione `.class` che verrà prodotto come bytecode dal compilatore Java sarà esattamente (con gli stessi caratteri maiuscoli e minuscoli) uguale al nome della classe definita nel file sorgente, indipendentemente da quello dato al file sorgente (che in ogni caso è bene abbia per chiarezza lo stesso nome).vedi#

Una classe di questo tipo compilata può essere usata come un'applicazione a sè stante interpretabile con la Java Virtual Machine, ma nulla esclude che possa essere anche utilizzata come classe particolare in un'altra applicazione.

Un'applicazione come quelle degli esempi fatti presenta i risultati sullo schermo in modalità testo, ma è anche possibile dare ad un'applicazione un'interfaccia grafica, vedremo come al paragrafo 4.4.7.

Generalmente l'ambiente di run-time non impone per un'applicazione le restrizioni imposte per ragioni di sicurezza alle applet come quelle relative al divieto di accesso al disco ed ai file della macchina client.

4.4.2 Esecuzione di un'applicazione

Per l'esecuzione di un'applicazione si adopera l'interprete Java (detto anche **Java Virtual Machine**), che nel JDK di Sun è un applicativo che si chiama proprio **`java`**, azionato con il

comando seguente, nel quale deve essere notato che il nome dell'applicazione viene scritto senza la sua estensione `.class`:

```
java miaApplicaz
```

Quando si lancia quest'istruzione in linea di comando l'interprete Java carica la classe e ne invoca automaticamente il metodo `main()`: essendo questo metodo `static`, e quindi non legato ad una particolare istanza della classe, non è necessario creare esplicitamente un oggetto della classe per eseguire l'applicazione.[DETTO#]

A proposito di questa dipendenza di un'applicazione Java dall'interprete (ed in pari modo di un'applet dall'interprete Java integrato nel browser) bisogna precisare che applicazioni ed applet Java non sono mai completamente indipendenti ma devono essere viste come integrate nell'ambiente di runtime che si incarica di invocarne certi metodi a seconda di determinate circostanze e anche di passare loro dei parametri (come si vedrà per esempio per l'oggetto di tipo `Graphics` passato al metodo `paint()` nelle applet).

Il programma Java compilato (cioè `miaApplicaz.class` nel caso dell'esempio) perché possa essere eseguito deve essere in una directory compresa nel "classpath" del sistema (vedi paragrafo 4.2.2).

Il link delle altre classi a quella `public` con un `main()` di partenza viene effettuato durante questa fase detta di run-time.

4.4.3 Passaggio di argomenti ad un programma Java

Per passare degli argomenti ad un programma Java (e più precisamente al suo metodo `main()` che prevede sempre come argomento un array di stringhe) li si scrivono di seguito al comando di esecuzione sulla linea di comando:

```
java miaApplicaz param1 param2 ...
```

dove `param1`, `param2`, etc., che sono separati tra di loro da almeno uno spazio, sono i valori dei parametri `args[0]`, `args[1]`, etc., da passare sotto forma di stringhe come argomenti nel caso che essi siano previsti nell'applicazione (si noti che c'è una differenza con il C e il C++, dove il posto del primo argomento era riservato al nome stesso dell'applicazione, prima voce della linea di comando, ed il primo argomento era `args[1]`).

Se un parametro è costituito da una stringa contenente degli spazi essa va racchiusa tra virgolette:

```
java miaApp param1 "param numero 2" param3
```

E' da notare che, siccome gli argomenti ad un programma Java sono sempre passati sotto forma di stringhe, anche quando rappresentano dei numeri, se si vogliono adoperare dei valori di tipo non `String` all'interno del programma bisogna poi convertirli, ad esempio per valori numerici di tipo intero con un'espressione:

```
Integer.parseInt(arg[n])
```

Esempio:#

4.5 Creazione ed esecuzione di un'applet

4.5.1 Creazione di un'applet

Un'applet deve essere rappresentata da una classe derivata per *subclassing* dalla classe `Applet` contenuta nel package di sistema `java.applet` (attenzione a minuscole e maiuscole!), che contiene in sé i comportamenti generali di un'applet e le sue possibili interazioni con il sistema, con l'utente e con il browser. La classe dell'applet deve essere necessariamente dichiarata `public` in quanto è il punto di partenza che deve essere accessibile dall'esterno, e può servirsi di altre classi (non necessariamente pubbliche) create in appoggio ad essa nel suo stesso file o in un package a parte importato.

Un esempio molto semplice di applet può essere il seguente:

```
// D07miaApplet.java (F.Spagna) Primo semplice esempio di applet

import java.awt.Graphics;

public class D07miaApplet extends java.applet.Applet {

   public void paint(Graphics g) {
      g.drawString("Ciao", 10, 20);
   }
}
```

che scrive la stringa `"Ciao"` nell'area dell'applet alla posizione 10 pixel a destra del lato sinistro e 20 pixel sotto il lato superiore, poiché, come vedremo bene al capitolo 7, l'origine delle coordinate (date in pixel) di ogni elemento da disegnare nell'applet è il vertice in alto a sinistra dell'area dell'applet, con x positive verso destra e y positive verso il basso.

Ecco il risultato prodotto da quest'applet sullo schermo nella finestra del browser:

#

Per quanto riguarda il metodo `paint(Graphics)` della classe `Applet` non ci soffermiamo per il momento su una sua spiegazione completa, ma diciamo qui soltanto che esso ha la funzione di disegnare l'applet da visualizzare sulla pagina che la contiene e viene messo in funzione automaticamente dal browser tutte le volte che la pagina deve essere ridisegnata in seguito ad operazioni fatte su di esso. Tutte le funzionalità di grafica dell'applet devono essere definite in questo metodo nelle sottoclassi create, perché nella classe `Applet` esso non disegna niente, ma il sistema sa quando esso deve andare in funzione. Il metodo deve essere `public` perché è `public` nelle superclasse `Applet` e quindi non può essere diverso nella sottoclasse. Esso riceve per argomento un oggetto della classe `Graphics` (vedi paragrafo 4.7.3.5) che potremmo definire semplicisticamente come lo strumento di disegno proprio dell'area dell'applet.

Una classe può essere un'applet ed un'applicazione allo stesso tempo, cioè essere una sottoclasse di `Applet` e presentare contemporaneamente un metodo `main()`: il suo comportamento dipenderà dal contesto nel quale essa sarà adoperata.

Quando in una pagina HTML si incontra il richiamo ad un'applet la classe dell'applet viene scaricata (*downloaded*) attraverso la rete dal server sotto forma di file .class compilato come bytecode e con essa anche tutte le altre classi che sono usate dall'applet. Contrariamente alle

applicazioni, per le quali il metodo main(), che è quello di partenza, non crea necessariamente un'istanza della classe dell'applicazione (lo fa solo se ne viene creata espressamente un'istanza nel main()), nel caso di un'applet viene sempre creata da parte del browser un'istanza della classe relativa, e se nella stessa pagina HTML ci sono varie applet con lo stesso nome, e magari con parametri diversi, vengono create altrettante istanze, ciascuna col suo comportamento.

4.5.2 Inserimento e visualizzazione di un'applet in un documento HTML

Un'applet compilata in uno o più file binari (uno per ogni classe) aventi l'estensione .class , tra i quali c'è quello che contiene la classe derivata da Applet (chiamato per esempio miaApplet.class), può essere richiamata in un file HTML *(HyperText Markup Language)* con estensione .html o .htm (situato, in una prima ipotesi, nella stessa directory del file .class) mediante il marcatore specifico HTML (*tag*) **<applet>**. Nella sua forma più semplice un file HTML che richiama un'applet si presenta nel seguente modo:

```
<! APPLET.HTML  F.Spagna  Esempio di applet richiamata in una pagina HTML>
<HTML>
<applet code="miaApplet.class">
</applet>
</HTML>
```

nella quale l'attributo:

- **code=**"nomeClasse.class" specifica il nome dell'applet compilata (compresa la sua estensione .class) che sarà visualizzata nel punto della pagina HTML in cui è richiamata con il suo *tag*, nome scritto rispettando esattamente i caratteri maiuscoli e minuscoli così come è stato definito nel codice sorgente e di conseguenza come esso figura nel nome del file compilato .class.

Tale file HTML richiede al server il bytecode dell'applet, lo carica sulla macchina client e manda in esecuzione l'applet se è visualizzato con un browser in cui è stato implementato l'interprete Java (Java Virtual Machine).

Se la directory in cui sono posti i file .class è diversa da quella del file HTML si deve precisare nel *tag* come attributo **codebase** il nome di quella directory relativa a quella del file HTML:

```
<applet code="nomeApplet"  codebase="nomeDirectory">
</applet>
```

in cui:

- **codebase=**"directory": dà il nome della directory in cui si trova l'applet, con il suo percorso (*path*) a partire dalla directory dell'HTML, ma quest'attributo può essere omesso se la classe si trova nella stessa directory del documento HTML, e può invece essere ampliato con un URL se viene fatto un richiamo ad un'applet posta in un altro server remoto, come ad esempio:

```
codebase="http://www.sitoRemoto.it/directory/"
```

Si tenga presente che nell'espressione precedente la barra / in fondo al nome dell'URL è obbligatoria.

Una versione di chiamata dell'applet da un documento HTML più completa di quelle esemplificate finora, contenente degli altri **attributi**, soprattutto le dimensioni dell'applet, può essere la seguente:

```
<applet code="miaApplet.class"  codebase="directory"
        width=pixelLarghezza  height=pixelAltezza
        align="top" (o "bottom" o "middle")
        . . . . . .
/applet>
```

Infatti al tag `<applet>` possono essere dati altri attributi, che sono gli stessi che possono essere assegnati ad un'immagine che fosse richiamata sull'HTML con il tag `<img>`. Tra essi ricordiamo:

- **width=**320 (ad esempio) e **height=**200 (ad esempio): assegnano le dimensioni (rispettivamente larghezza ed altezza) in pixel dell'area rettangolare riservata all'applet sulla pagina (quest'area sarà sempre ben definita e visibile sulla pagina, anche se l'applet non fosse disponibile o per qualche ragione non funzionasse). Nella programmazione dell'applet bisognerà tenere presenti le dimensioni che essa avrà sullo schermo per le funzioni di disegno o di posizionamento di elementi in modo da non uscire dai suoi margini.

- **align=**"top" (ad esempio): si riferisce alla posizione verticale dell'area dell'applet rispetto al rigo di scrittura dei caratteri del testo HTML: con **left** e **right** l'applet è posta sulla riga corrente rispettivamente sul lato sinistro e destro della pagina sullo schermo ed il testo continua su righe nell'altro lato della pagina; con **texttop**, **middle** e **bottom** (o **baseline**) si ha un allineamento verticale dell'area dell'applet con la riga di testo rispettivamente all'altezza del bordo superiore, della linea mediana orizzontale e del bordo inferiore. Altri termini possibili sono **top**, **absmiddle**, **absbottom**. Si tenga presente che l'area dell'applet, così come è per le immagini, viene posizionata nella pagina nella stessa posizione in cui sarebbe inserito il prossimo carattere se anzichè l'applet continuasse il testo.

Nel caso in cui l'applet è contenuta in un file di tipo JAR (vedi paragrafo 4.10) compresso, oltre all'attributo `code=` che deve ancora essere presente, anche l'attributo:

```
archive="JarFileName.jar"
```

Altri possibili parametri dell'applet a livello HTML associati a `left` e `right` sono **hspace**, che fissa in pixel la distanza orizzontale lasciata tra l'area dell'applet ed il testo circostante del documento, e **vspace** quella verticale tra la riga iniziale del testo ed il margine superiore dell'applet. Esiste anche un tag **alt** che viene usato per il caso in cui il browser riconoscesse l'applet ma non riuscisse ad eseguirla.

Ecco qui di seguito un esempio che mostra insieme i vari attributi del tag <applet>:

```
<APPLET
  CODE = nome del file dell'applet
  [CODEBASE = url dell'applet]
  WIDTH = larghezza in pixel
```

```
HEIGHT = altezza in pixel
[OBJECT = nome del file contenente l'eventuale applet serializzata]
[ARCHIVE = lista di file jar]
[ALT = testo alternativo]
[NAME = nome dell'applet]
[ALIGN = allineamento dell'area dell'applet]
[VSPACE = spazio verticale in pixel tra applet e testo]
[HSPACE = spazio orizzontale in pixel tra applet e testo]
>
[<PARAM NAME = nome parametro1 VALUE = valore parametro1 (stringa)>]
[<PARAM NAME = nome parametro2 VALUE = valore parametro2 (stringa)>]
 . . .
[testo alternativo]
</APPLET>
```

Presentiamo un esempio:#

```
<APPLET
  CODE = nome del file dell'applet
  [CODEBASE = url dell'applet]
  WIDTH = larghezza in pixel
  HEIGHT = altezza in pixel
  [OBJECT = nome del file contenente l'eventuale applet serializzata]
  [ARCHIVE = lista di file jar]
  [ALT = testo alternativo]
  [NAME = nome dell'applet]
  [ALIGN = allineamento dell'area dell'applet]
  [VSPACE = spazio verticale in pixel tra applet e testo]
  [HSPACE = spazio orizzontale in pixel tra applet e testo]
>
[<PARAM NAME = nome parametro1 VALUE = valore parametro1 (stringa)>]
[<PARAM NAME = nome parametro2 VALUE = valore parametro2 (stringa)>]
 . . .
[testo alternativo]
</APPLET>
```

In figura 4.1 è mostrato l'effetto sulla pagina del browser.

Figura 4.1 Esempio di posizione di un'applet in una pagina HTML.

4.5.3 Applet e package importati

Se un'applet, il cui bytecode (cioè il file compilato .class) è posto in una certa directory del server, importa (con un'istruzione di `import`) un particolare package non compreso tra i package di sistema di Java, si dovrà creare nella directory dell'applet una sottodirectory portante lo stesso nome del package, nella quale saranno poste le classi compilate appartenenti al package stesso.

Per fissare le idee facciamo l'esempio di un server che ha in una sua directory `htdocs` un documento HTML chiamato `doc.html` che richiama un'applet posta in una sottodirectory

htdocs/classi (ma l'applet potrebbe anche essere nella stessa directory dell'HTML) e che quest'applet, chiamata appl, richiama nel codice direttamente un certo numero di classi cla1, cla2, cla3, e importa altre classi tramite il package che esse costituiscono, chiamato, diciamo, msql. La disposizione dei vari file sul server sarà quella mostrata nello schema seguente (diciamo come una curiosità che lo schema è stato da noi disegnato con un'applicazione Java).

#

4.5.4 Passaggio di parametri dall'HTML ad un'applet

Oltre ai suddetti attributi dell'applet, possono essere inseriti nel documento HTML anche altri attributi per passare dei parametri dall'HTML all'applet nel caso in cui il suo codice preveda di ricevere degli argomenti dall'esterno:

```
<applet code="nomeClasse.class">
      <param name="parametro1"  value="stringa1">
      <param name="parametro2"  value="stringa2">
          . . . . . . . . . .
      <param name="parametroN"  value="stringaN">
</applet>
```

Ogni parametro viene introdotto mediante due attributi: il nome con cui il parametro sarà riconosciuto a livello del codice del programma Java ed il valore assegnato ad esso sempre sotto forma di una stringa passata all'applet. La forma è la seguente:

- **param name=**parametro1 (ad esempio) dove parametro1 è il nome del parametro come compare nel programma (scritto rispettando le maiuscole e le minuscole) (VERO?#) cui si assegna il valore dato dal successivo value=;

- **value=**stringa1 (ad esempio) dove stringa1 è la stringa (non è strettamente necessario rinchiuderla entro virgolette " "VEDI#) che rappresenta a caratteri il valore del parametro indicato in param name= , passato all'applet come argomento, avendo previsto nel codice dell'applet (di solito nel suo metodo init()) la lettura dall'HTML di ciascun parametro con l'utilizzazione di volta in volta del metodo getParameter(String):

```
String p1 = getParameter("parametro1");
```

cui si dà il nome del parametro come argomento e che restituisce il valore letto come stringa dall'HTML. Nel caso si vogliano passare dei valori numerici questi vengono messi sotto forma di stringhe che poi vanno convertite in numeri nel programma.

Un esempio potrebbe essere il seguente:

```
<applet code=miaClasse.class   codebase=http://www.mioSito.it/dir/
          align=top
          width=320  height=200
          param name=Nominativo   value=Spagna
          param name=Matricola  value=79960
 /applet>
```

che presuppone nel codice dell'applet, generalmente nel metodo `init()`, la presenza della richiesta al documento HTML dei due parametri `Nominativo` e `Matricola` rispettando i caratteri minuscoli e maiuscoli, mediante il metodo `getParameter(String)` usato in istruzioni del tipo:

```
String nomin = getParameter("Nominativo");
String matricola = getParameter("Matricola");
int matr = Integer.parseInt(matricola);
```

con la trasformazione da stringa a intero mediante il metodo `parseInt(String)` della classe `Integer` che gestisce gli interi come oggetti, fornendo loro varie funzionalità.

Se un parametro atteso non è specificato o non è stato scritto in modo corretto nel documento HTML, il metodo `getParameter()` restituisce `null`. Un controllo su questo valore, che è buona norma effettuare nel codice dell'applet per il caso in cui risultasse di valore `null`, permette eventualmente di predisporre un valore di default del parametro nel codice dell'applet come segue:

```
String nomin = getParameter("Nominativo");
if (nomin == null) nomin = "ValoreDefault";
```

Esiste un altro attributo per le applet chiamato **name** che serve per individuarla sulla pagina HTML, per esempio quando si ha a che fare con diverse applet contemporaneamente presenti sulla stessa pagina che devono comunicare tra di loro (di questo si parlerà nel paragrafo 4.5.11 dedicato a questo argomento), o per l'interazione tra Javascript e applet (vedi paragrafo 4.5.12).

4.5.5 Processo dell'esecuzione di un'applet in un browser

Il processo seguito da un browser (per esempio HotJava) in relazione al caricamento ed all'esecuzione di un'applet inserita in un documento HTML è il seguente:

- il cliente richiede un documento HTML ad un server, scrivendo un URL nella linea di comando del suo browser;
- il browser analizza la richiesta e la passa al gestore di protocollo di rete, che si rivolge al server per trasmettergli la richiesta;
- il server invia il documento HTML richiesto al cliente, il cui browser lo analizza;
- se nel documento HTML il browser trova un tag `<applet>` ne richiede il bytecode al server;
- il server invia il bytecode al cliente e il browser lo sottopone alle sue specifiche verifiche di sicurezza;
- se il bytecode supera i controlli di sicurezza viene caricato in memoria dal sistema di run-time Java del browser;
- l'interprete Java del browser istanzia l'applet e ne esegue il metodo `init()`;
- il browser visualizza il documento HTML ed esegue il metodo `start()` che fa partire l'applet ed il metodo `paint()` che la visualizza;

- se il cliente rimuove il documento HTML il browser esegue il metodo `stop()` dell'applet;
- il browser esegue infine il metodo `destroy()` e segnala la fine al sistema di run-time;
- il sistema di run-time libera la memoria che era stata occupata dall'applet.

4.5.6 Visualizzazione ed esecuzione di un'applet senza un browser

Un'applet inserita in un documento HTML risiedente con esso su una macchina locale può essere visualizzata anche senza il browser direttamente con un'applicazione specifica chiamata **appletviewer** mediante il comando:

```
appletviewer nomeFile.html
```

dove `nomeFile.html` è il nome del documento HTML contenente l'applet.

E' possibile visualizzare ed eseguire un'applet senza il browser mediante l'applicativo **`appletviewer`** che viene fornito con il JDK stesso, che visualizza un file HTML contenente l'applet. Il comando è:

```
appletviewer miaApplet.html
```

ma un'applet può anche essere vista creando un'applicazione Java indipendente che prepara la finestra in cui l'applet può essere visualizzata e fatta funzionare (quindi facendo quello che fa normalmente il browser) e questo sarà visto al paragrafo 4.5.10.

4.5.7 Serializzazione delle applet

Le applet possono essere serializzate in un file. Un'applet di questo tipo quando viene deserializzata parte con il metodo `start()` e non con `init()` in quanto quest'ultimo metodo è già stato eseguito all'inizio del ciclo di vita dell'applet e quindi prima della serializzazione, mentre il metodo `start()` prima della serializzazione o non è ancora stato invocato o, se è stato invocato, deve essere seguito da uno `stop()` perchè l'applet possa essere serializzata: infatti al momento della serializzazione l'applet deve essere già inizializzata ma ferma.

Un'applet serializzata viene specificata con l'attributo `OBJECT` al posto di `CODE` nel tag HTML `<APPLET>`.

4.5.8 Applet in file JAR

Le applet possono essere contenute in un file di tipo JAR (vedi paragrafo 4.10), che può anche contenere, oltre all'applet stessa, anche altri file come altri file .class delle classi accessorie dell'applet, immagini, suoni ed ogni altra cosa che può servire all'applet stessa. Le applet di questo tipo sono specificate con l'attributo `ARCHIVE` nel tag HTML `<APPLET>`.

4.5.9 Creazione di un'applicazione Java indipendente in modalità grafica (AWT)

Alle applicazioni Java può essere data un'interfaccia grafica. Perchè un'applicazione grafica possa essere visibile sullo schermo deve essere contenuta in un `Frame` (cioè un componente grafico di tipo finestra) in cui possono essere posti i suoi componenti: è necessario quindi creare un'istanza di `Frame` dell'AWT, che possiede le funzionalità di disegno (con `paint()`), di contenitore di altri componenti AWT (component aggiunti con `add()`) e di risposta agli eventi (metodi event handler). A questo riguardo si offrono due possibilità, che sono sostanzialmente equivalenti: o si definisce e istanzia un oggetto di tipo `Frame` al suo interno sul quale si agisce, oppure è l'applicazione stessa che viene definita come sottoclasse di `Frame` derivando la classe dell'applicazione come sottoclasse dalla classe `Frame` del package `java.awt`, o da una sottoclasse di questa, e che quindi viene usata come tale. In ogni caso il `Frame` può contenere componenti come `Canvas`, `Panel` o anche `Applet` (che sono in fondo dei `Panel`). Ecco i due esempi fatti utilizzando i due sistemi:

```
// D08appFram.java (F.Spagna) Esempio di applicazione grafica Java con Frame

import java.awt.*;                                          // per Frame e Label

public class D08appcFram {          // applicazione in cui si istanzia un Frame

   public static void main(String s[]) {                  // avvia l'applicazione

      Frame fr = new Frame("Titolo");                 // crea una finestra tipo Frame
      fr.resize(300, 200);                                 // dimensioni finestra
      fr.setFont(new Font("Arial", Font.BOLD, 24));              // font nel Frame
      fr.add(new Label("buongiorno"));                     // componente interno
      fr.show();                                        // finestra resa visibile
   }
}
```

Secondo esempio:

```
// D09appFram.java  F.Spagna  Esempio di applicazione grafica Java come Frame

import java.awt.*;                                          // per Frame e Label

public class D09appFram extends Frame {  //applicaz. come sottoclasse di Frame
                                                 // cui si aggiunge un main()
   public static void main(String s[]) {                  // avvia l'applicazione

      appFram aFr = new appFram("Titolo");                   // crea una finestra
//    aFr.setTitle("Titolo");                                 // titolo finestra
      aFr.resize(300, 200);                                // dimensioni finestra
      aFr.setFont(new Font("Arial", Font.BOLD, 24));             // font nel Frame
      aFr.add(new Label("buongiorno"));                    // componente interno
      aFr.show();                                       // finestra resa visibile
   }
}
```

Il risultato è in ambedue i casi quello presentato in figura 4.2
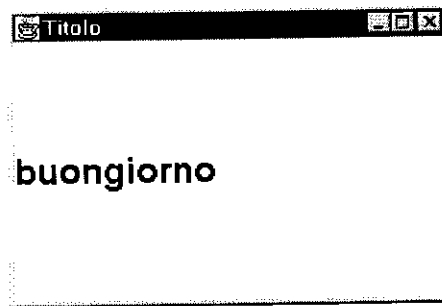
Figura 4.2 Frame di un'applicazione Java

Un'applicazione Java in modalità grafica può essere creata derivandola dalla classe `Frame` con il seguente codice:serve?#

```
// D10appGraf.java (F.Spagna) Esempio di applicazione Java

import java.awt.*;

public class D10appGraf extends Frame {

   public static void main(String args[]) {
      new appGraf();                              // istanzia un oggetto appGraf
   }
   public appGraf() {
      super("Applicazione Java");                             // titolo del Frame
      setSize(320, 400);                  // stabilisce le dimensioni del Frame
      setVisible(true);                              // da' la visibilita' al Frame
   }
   public void paint(Graphics g) {
      g.setColor(Color.red);
      g.fillRect(10, 10, 90, 90);
   }
   public boolean handleEvent(Event e) {
      if (e.id == Event.WINDOW_DESTROY)
         dispose();
      return super.handleEvent(e);
   }
}
```

4.5.10 Applicazione Java da un'applet inserita in un `Frame`

Ma un'applicazione Java indipendente può anche essere creata a partire da un'applet che viene fatta funzionare in una finestra di tipo Frame anziché nella finestra del browser.

Se ad esempio abbiamo già creato un'applet chiamata `miaApplet` si può inserirla in un altro programma così concepito:

public class ospitaApplet extends Frame

(v. pag.268-269 libro Buss)#

poi altro programma indipendente p.264 libro Buss#

Sia che il `Frame` sia creato in uno o nell'altro dei due modi del paragrafo precedente#, se il programma è un'applicazione ed un'applet allo stesso tempo si dovrà istanziare l'`Applet` e quindi aggiungerla al `Frame` con `add()` come si farebbe per un pannello qualsiasi, dopo di che si invocano i metodi `init()` e poi `start()` dell'applet, e tutto ciò è fatto nell'esempio che segue:

```
// D1lappletApp.java (F.Spagna) Esempio di applicazione grafica con un'Applet

import java.awt.*;

public class D1lappletApp extends java.applet.Applet {

   public static void main(String s[]) {            // avvia l'applicazione

      Frame fr = new Frame("Titolo");        // crea una finestra tipo Frame

                                                      // crea un'applet
      appletApp ap = new appletApp();
      ap.setFont(new Font("Arial", Font.BOLD, 24));    // font nell'applet
      ap.add(new Label("buongiorno"));     // componente interno all'applet

                                                   // dimensioni finestra
      fr.resize(300, 200);
      fr.add(ap);                              // aggiunge l'applet al Frame
      fr.show();                                  // finestra resa visibile

                                                 // inizializza l'applet
      ap.init();
      ap.start();                                  // fa partire l'applet
   }
}
```

Il risultato è presentato in figura 4.3 ed è simile a quello della figura precedente salvo per la posizione della scritta che adesso è al centro, in quanto il layout è quello di default dell'applet (cioè il `FlowLayout`), mentre per il `Frame` era il `BorderLayout`.
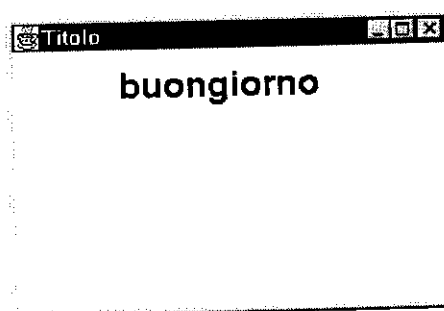
Figura 4.2 Applet Java vista come applicazione

Se un'applet cosiffatta facesse ricorso ad un `AppletContext` si produrrebbe una `NullPointerException`, e ciò per esempio avviene quando si invocano i metodi `getDocumentBase()`, `getParameter()` o `getImage()`. In questi casi è tuttavia ancora a volte possibile ricorrere a quei metodi su un istanza della classe `Toolkit` ottenuta con il metodo statico di classe `Toolkit.getDefaultToolkit()`. Questo è possibile per

getImage(), ma non per getAudioClip(), per i quali sono disponibili invece delle classi nel package sun.audio.

Esempio# (immagini e poi audio in applicazione non grafica):

Un'applicazione Java indipendente (cioè non legata ad una pagina HTML e ad un browser) può funzionare in modalità grafica utilizzando componenti AWT. Per questo l'applicazione deve essere derivata come sottoclasse della classe Applet (ed è effettivamente un'applet) ereditandone i metodi di funzionamento grafico, ma deve avere anche un metodo main(String args[]).

Un'applicazione Java indipendente con un'interfaccia grafica può anche essere creata a partire da un'applet: per far ciò si aggiunge all'applet un metodo main() in cui si istanzia un Frame che conterrà l'applet, si crea un'istanza dell'applet stessa, la si aggiunge al Frame e la si fa partire con i metodi init() e start() chiamati in questo caso esplicitamente (per le applet invece è il browser ad invocare automaticamente questi metodi) e infine si visualizza il frame con show(). Facendo in questo modo si crea un Frame in cui si fa funzionare l'applet stessa facendola partire come farebbe il browser.

Esempio:

```
// D12appInFram.java (F.Spagna) Applet inserita in un Frame

import java.awt.*;

public class D12applInFram extends java.applet.Applet {

   public void init() {
                                                   // dimensiona l'applet
      resize(100, 100);
   }
   public void paint(Graphics g) {
                                                   // disegnino su applet
      g.fillRect(20, 20, 30, 20);
      g.drawRect(0, 0, size().width-1, size().height-1);
   }                                         // disegna la cornice dell'applet
   public static void main(String s[]) {
                                                       // titolo Frame
      Frame f = new Frame("Frame con applet");      // dimensiona il Frame
      f.setSize(300, 200);                   // adatta le dimensioni del Frame
      //f.pack();                                // rende il Frame visibile
      f.show();
                                                   // istanzia un'applet
      applInFram ap = new applInFram();           // la aggiunge al Frame
      f.add(ap);                                  // inizializza l'applet
      ap.init();                                        // avvia l'applet
      ap.start();
   }
}
```

Un'applicazione così creata può funzionare da applicazione o da applet a seconda del contesto nel quale è adoperata. Infatti il programma precedente compilato con:

```
javac applInFram.java
```

porta alla creazione di un file `applInFram.class` che può essere usato per la visualizzazione dell'applet in due modi: direttamente come applet con:

```
javac applInFram.java
```

se il file `applInFram.html` contiene il tag:

```
<applet code=applInFram.class width=600 height=400></applet>
```

oppure come applicazione Java con:

```
javac applInFram.java
```

Un'applet fatta funzionare in queste condizioni sia con il primo sia con il secondo modo non è soggetta alle retrizioni relative all'accesso ai dati del sistema che sono imposte alle applet dai browser e può pertanto ad esempio leggere e scrivere sul sistema locale (ad esempio con istruzioni poste nel suo metodo `init()`, o nel caso di applicazione anche nel metodo `main()`). Con l'appletviewer comunque qualunque applet può sempre fare queste operazioni, non passando da un browser.

Esempio pag.288 di Lemay#

4.5.11 Comunicazione tra applet

Diverse applet possono essere inserite sulla stessa pagina HTML e possono comunicare tra di loro purchè a ciascuna applet sia assegnato un proprio nome mediante l'attributo `NAME` del tag `<APPLET>` in modo che possa essere individuata dal browser:

```
<APPLET CODE=appl.class NAME=nomeApplet1></APPLET>
```

Un programma complesso sviluppato in un'applet potrebbe essere tale da produrre un bytecode così grande da portare a tempi proibitivi di download dal server, oltre a comportare problemi per la manutenzione, e può allora essere conveniente spezzare il programma in più applet di dimensioni ragionevoli e farle comunicare tra di loro.

La comunicazione diretta tra diverse applet appartenenti alla stessa pagina HTML é prevista dalla libreria standard di Java. Se però le diverse applet sono poste su pagine o *frame* HTML diversi la comunicazione diventa più complessa. Una tecnica per risolvere questo problema é stata proposta da A.Meckler in [4.4.8] ed è basata sulla creazione di una classe comune accessibile dalle diverse applet, che funzioni da centrale di comunicazione tra di esse.

[4.4.8] Andrew Meckler, Java and Inter-Applet Communication, Dr. Dobb's Journal #270, October 1997, page 46.

Il metodo `getApplets()`

`getAppletContext().getApplets(nome)`

Restituisce un'`Enumeration`

HasMoreElements

NextElement

Send Message

Esempio pari pari di Lemay di pag. 299#

Restituisce un'applet da un'altra applet sulla stessa pagina.

L'applet allora può essere trattata come qualsiasi altro componente.

4.5.12 Comunicazione tra HTML e applet

I metodi pubblici di un'applet presente in una pagina HTML possono essere invocati in risposta agli eventi di un form presente sulla stessa pagina. Perché questo sia possibile all'applet deve essere assegnato un nome ed i suoi metodi sono invocati facendoli precedere dal nome dell'applet, ad esempio in seguito ad un evento `ONCLICK` di un `BUTTON`. Ecco un esempio, in cui una pagina HTML contiene un'applet contenente una rotella che gira e che può essere fermata o fatta ripartire con due bottoni presenti sulla stessa pagina non facenti parte dell'applet ma di un form HTML:

```
<! ruota.html  F.Spagna  Esempio di comando di un'applet
                         da un bottone esterno all'applet>
<HTML>

<APPLET
    CODE=ruota001.class HEIGHT=100 WIDTH=120 NAME=ruota>
</APPLET>

<FORM>
   <INPUT TYPE=BUTTON ONCLICK=ruota.stop()  VALUE="ferma">
   <INPUT TYPE=BUTTON ONCLICK=ruota.start() VALUE="parti">
</FORM>

</HTML>
```

In figura 4.1 è riportata l'applet ed i due bottoni di comando non facenti parte dell'applet, ma comunicanti con essa attraverso il Javascript.
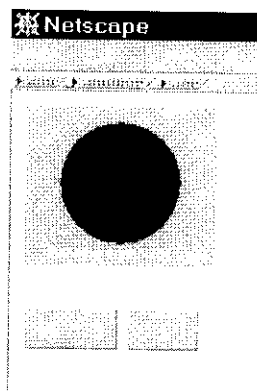
Figura 4.1 Un'applet comunicante con due bottoni esterni ad essa.

L'esempio è della massima semplicità, ma combinando opportunamente il Javascript e le applet si possono fare operazioni anche molto elaborate.

Esempio più complesso con altri metodi.#

4.5.13 Comunicazione tra Javascript e applet

Java e Javascript giocano un ruolo diverso sulle pagine HTML: mentre le applet Java sono confinate in una loro propria area delimitata sulla pagina entro la quale soltanto possono interagire con l'utente, il Javascript permette invece di creare degli script che rispondono agli eventi generati sull'intera pagina, la sua finestra e tutti gli oggetti in essa contenuti, e anche creare altre finestre ed interagire con esse. Questi due ruoli possono essere integrati creando applet che funzionano come componenti della pagina e collegandole mediante il Javascript e questo è possibile perché il Javascript può comunicare con le applet presenti sulla pagina, accedendo alle loro proprietà e metodi ed infatti i metodi pubblici dell'applet possono essere invocati dal Javascript. Perché questo sia possibile all'applet deve essere assegnato un nome ed i suoi metodi sono invocati facendoli precedere dal nome dell'applet.

[4.x] Jamie Jaworsky, Scripting Java applets, http://www.builder.com/Programming/Scripter/120998/, 12/9/98.

4.6 Chiamata di un programma CGI da un'applet

Un'applet può chiamare all'esecuzione in modalità CGI un programma su un server Web HTTP simulando esattamente il flusso di dati che di norma viene inviato quando si preme il bottone "submit" di un form HTML ed inviandolo al server. Con richiesta di tipo GET basta inviare al server una stringa URL in cui sono codificati i dati del form.

Si crea un URL con:

```
http://www.server.it/cgi-bin/script.exe?a=1&b=2
```

e poi lo si passa al browser col metodo `showDocument()`:

```
GetAppletContext.showDocument(new URL(...))
```

Esempio:#

4.7 Lettura e scrittura di file in un'applet

Per aggirare il criterio di sicurezza delle applet che vieta loro la possibilità di leggere o scrivere file sul disco del sistema locale, nel caso ci sia bisogno di conservare i dati prodotti da

un'applet si può ricorrere a file posti sul server da cui è partita l'applet, dal momento che su di esso le applet sono autorizzate ad accedere alle risorse di sistema. Una possibilità del genere viene offerta al sito http://www.focus-asia.com/java/jfs con il protocollo di sistema di file di rete di tipo TCP/IP detto JFS, che permette di caricare e salvare file su un server remoto. Un server JFS funziona in accoppiamento con un Web server su un sistema Unix e le applet JFS stabiliscono una connessione ad esso e possono scrivere e leggere file posti in una sottodirectory del server. Le comunicazioni tra client e server avvengono tramite `request` (dal client) e `reply` (dal server). Per esempio per caricare un file dal file system JFS del server il client invia una richiesta `Get` e il server risponde con un `Data` o un `Fail`.

Un'applet che voglia comunicare con un server JFS deve usare la classe `JFSclient` i cui metodi sono disponibili per richieste di azioni sui file del server. In particolare:

```
byte[] get(String path, int ver)
```

provvede al caricamento di un file dal server e:

```
void put(String path, int ver, byte[] data, String type)
```

provvede al salvataggio di un file sul server, mentre il costruttore:

```
JFSclient(String host)
```

apre la connessione con un server JFS. Tra gli altri metodi notiamo, tralasciando per semplicità gli argomenti:

```
getdir(), delete(), mkdir(), copy(), rename()
```

i cui nomi hanno un significato chiaro.

4.8 Classe Applet del package `java.applet`

4.8.1 Variabili d'istanza della classe `Applet`

Dato il suo ruolo fondamentale, la **classe `Applet`** del package `java.applet` viene descritta in maniera dettagliata in questo paragrafo e nel successivo.

Le **variabili d'istanza** della classe `Applet`, tutte di tipo **`public`**, sono:

`URL appletURL`	URL dell'applet
`URL documentURL`	URL del documento HTML contenente l'applet
`Color bgColor`	colore di sfondo dell'applet
`Color fgColor`	colore di primo piano dell'applet
`int width`	larghezza in pixel dell'area rettangolare occupata dall'applet
`int height`	altezza in pixel dell'area rettangolare occupata dall'applet
`Font font`	tipo di carattere usato nelle scritte entro l'applet
`TagRef tag`	tag HTML richiesto per chiamare l'applet
`AppletDisplayItem item`	nome dell'area (rettangolare) occupata dall'applet

I metodi

4.8.2 Caratteristiche di un'applet

Un'applet ha le seguenti caratteristiche, che per esempio in un ambiente di sviluppo come Visual Café di Symantec possono essere fissate su una tabella apposita:

nome	(esempio: `miaApplet`)
colore di fondo	(esempio: `255, 255, 255`)
colore del testo	(esempio: `0, 0, 0`)
larghezza in pixel	(esempio: `300`)
altezza in pixel	(esempio: `200`)
font: nome	(esempio: `Dialog`)
dimensione	(esempio: `12`)
style bold	(si/no)
italic	(si/no)
cursore	(esempio: `DEFAULT_COURSOR`)
layout	(esempio: `BorderLayout`)
abilitata	(si/no)
visibile	(si/no)
eredita colori	(si/no)
eredita font	(si/no)

Ecco come si presenta la tabella nell'ambiente di sviluppo suddetto:

Background	white
Bounds	
X	0
Y	0
Width	300
Height	200
Class	java.applet.Applet
Cursor	DEFAULT_CURSOR
Enabled	true
Font	
Name	Dialog
Size	12
Style	
Bold	false
Italic	false
Foreground	black
Inherit Background	false
Inherit Font	true
Inherit Foreground	true
Layout	BorderLayout
Name	miaApplet
Visible	true

Figura 4.2 Proprietà di un'applet da fissare in fase di progetto in Visual Café.

4.8.3 Metodi `public` della classe `Applet`

I metodi di tipo **`public`** della classe `Applet` sono riportati di seguito. Si noti che sempre dove è richiesto un nome di file di tipo `String`, deve essere messo l'indirizzo

completo se il file è in un'altra directory, e che vi deve anche essere messo un indirizzo URL con la relativa directory se si vuole fare riferimento a un file remoto.

	Applet()	metodo costruttore dell'applet
String	**getAttribute**(String)	ricava un attributo (parametro) dal documento HTML
Color	**getColor**(int,int,int)	restituisce il colore corrispondente a una terna di valori RGB
void	**getFocus**()	attiva l'applet
Font	**getFont**(String, int)	restituisce il font relativo ad un nome ed una dimensione dati
Font	**getFont**(String,int,int)	restituisce il font relativo a nome, stile e dimensione dati
Image	**getImage**(String)	restituisce l'immagine relativa ad un dato nome di file
Image	**getImage**(URL)	restituisce l'immagine relativa ad un dato URL
void	**gotFocus**()	dice se l'applet è stata attivata (?)#
boolean	**isActive**()	dice se l'applet è attiva nel documento HTML
void	**keyDown**(int)	chiamato se premuto un tasto, valore di questo passato
void	**lostFocus**()	dice se l'applet è stata disattivata (?)#
void	**mouseDown**(int, int)	chiamato se bottone mouse premuto, valori (x,y) passati
void	**mouseDrag**(int, int)	chiamato se mouse trascinante, valori (x,y) passati
void	**mouseEnter**(int, int)	chiamato se il cursore del mouse entra nell'area dell'applet
void	**mouseExit**(int, int)	chiamato se il cursore del mouse esce dall'area dell'applet
void	**mouseMove**(int, int)	chiamato se il cursore del mouse mosso, valori (x,y) passati
void	**mouseUp**(int, int)	richiamato se bottone mouse rilasciato, valori (x,y) passati
void	**paint**(Graphics)	richiamato per visualizzare l'applet nella sua area
void	**repaint**()	l'applet viene ridisegnata nella sua area
void	**play**(String)	riproduce suono di un oggetto AudioData dato col nome
void	**play**(AudioData)	riproduce suono di un dato oggetto AudioData
void	**resize**(int, int)	modifica le dimensioni dell'area rettangolare dell'applet
void	**showDocument**(URL)	visualizza in una finestra a parte la pagina di un URL
void	**showStatus**(String)	scrive una stringa nella linea di stato del browser
void	**update**(Graphics)	aggiorna la grafica dell'area dell'applet
void	**startPlaying**(InputStream)	inizia la ripoduzione di un flusso di dati audio
void	**stopPlaying**(InputStream)	interrompe la riproduzione di un flusso di dati audio
AudioData	**getAudioData**(String)	dà un oggetto AudioData data la stringa del nome
AudioData	**getAudioData**(URL)	dà un oggetto AudioData dato l'URL in cui si trova
InputStream	**getAudioStream**(URL)	dà il flusso di dati audio relativo ad un dato URL
InputStream	**getContinuousAudioStream**(URL)	dà il flusso continuo di dati audio di un dato URL

Qui?#

void	**removeAll**()	cancella gli elementi aggiunti all'applet con add()

4.8.4 Principali metodi della classe `Applet` richiamati dal browser

Le applet, che sono ospitate su una pagina HTML e su di essa eseguite, interagiscono con il browser che si prende carico di loro e, in relazione a diversi eventi, come inizializzazione, partenza dell'applet, sua fermata, chiusura dell'applet e disegno sullo schermo, invoca particolari specifici metodi (`public`) dell'applet stessa. Questi metodi non sono chiamati esplicitamente nel codice dell'applet, ma rappresentano dei punti di comando dall'esterno che è il browser a manovrare. Infatti è il browser che istanzia l'applet e ne dispone come se fosse un suo oggetto, del quale può invocare i metodi a sua discrezione attraverso messaggi che esso invia all'applet, in relazione ad eventi che si manifestano nell'ambiente di visualizzazione dell'HTML secondo lo schema:

browser => JVM => eventi => metodi pubblici => applet

Dei metodi `public` della classe `Applet` di particolare rilievo perchè sono attivati automaticamente in risposta ad eventi importanti e quindi controllano il funzionamento dell'applet stessa, sono **`init()`**, **`start()`**, **`stop()`**, **`destroy()`** e **`paint()`**. Questi metodi in partenza, cioè così come sono ereditati dalla classe `Applet`, sono vuoti, cioè non prevedono nessuna azione, ma il più delle volte sono ridefiniti dal programmatore (ma non necessariamente tutti) per rispondere ai comportamenti propri dell'applicazione.

Data la loro importanza, dopo aver ricordato che sono tutti `public` (e `void`), vediamoli un po' più da vicino uno per uno nei paragrafi seguenti.

4.8.4.1 Inizializzazione e metodo `init()`

Il metodo **`init()`** viene richiamato automaticamente dal browser nel momento in cui si crea un'istanza dell'applet, ossia quando l'applet (cioè il suo bytecode) è caricata per la prima volta in memoria, e prima della sua esecuzione. Il browser Netscape lo richiama anche quando l'applet è ricaricata o si torna ad una pagina che la contiene da cui si era usciti.

In questo metodo, come indica anche il suo nome, generalmente vengono introdotte operazioni di inizializzazione dei dati dell'applet, di creazione di oggetti che l'applet utilizza,, di caricamento di immagini o altre operazioni necessarie all'inizio dell'esecuzione. Di solito è in questo metodo che viene fatto anche il dimensionamento dell'area dell'applet con il metodo `resize(int,int)` e la lettura dei parametri e degli attributi passati all'applet dal documento HTML mediante i metodi `getAttribute(String)` e `getParameter(String)`.

4.8.4.2 Partenza e metodo `start()`

Il metodo **`start()`** viene richiamato automaticamente dal browser quando viene fatta partire l'applet, cioè ogni volta che il documento HTML cui l'applet appartiene viene reso visibile, ossia la prima volta e ogni volta che si ritorna alla pagina dopo una sua momentanea perdita di visibilità (per aver lasciato il posto ad un altro documento o per un ridimensionamento della finestra). La prima volta questo metodo è chiamato dal sistema subito dopo il metodo `init()`.

In questo metodo vengono fatte le operazioni di inizializzazione proprie del riavvio dell'applet, che sono in generale diverse da quelle del metodo `init()` di inizializzazione al

momento del caricamento dell'applet. Se l'applet prevede l'esecuzione in un thread è in questo metodo che si crea e si fa partire il thread con il suo `start()`, che farà partire il `run()`.

Durante la vita di un'applet il metodo `start()` può essere eseguito più di una volta, mentre il metodo `init()` è eseguito una volta sola.

4.8.4.3 Fermata e metodo `stop()`

Il metodo **`stop()`** viene richiamato automaticamente dal browser quando l'applet viene fermata e cioè ogni volta che viene tolto momentaneamente dalla visualizzazione nel browser il documento HTML contenente l'applet perchè si sostituisce alla pagina in corso un altro documento o perchè la finestra del documento è ridimensionata. In questi casi quasi sempre si deve prevedere in questo metodo l'arresto di ogni attività dell'applet chiudendo tutti i thread attivi, che altrimenti continuano comunque ad agire anche senza la presenza dell'applet sullo schermo, per non occupare più inutilmente il processore e in generale le risorse del sistema, anche se si può presentare qualche caso particolare in cui si può invece volere proprio che in quelle condizioni una certa azione continui. I thread eventualmente fermati in questo metodo possono poi essere fatti ripartire con il metodo `start()`.

Quando si definisce il metodo `start()` di solito si ridefinisce corrispondentemente anche il metodo `stop()`.

Questo metodo precede sempre l'esecuzione del metodo `destroy()`.

4.8.4.4 Distruzione e metodo `destroy()`

Il metodo **`destroy()`** viene richiamato automaticamente dal browser prima che l'applet venga completamente scaricata dalla memoria quando si chiude il browser o quando l'applet viene ricaricata in memoria, e può essere usato per terminare in modo sicuro l'esecuzione dell'applet e per liberare la memoria occupata da essa e da ogni altro oggetto connesso. Vi si possono prevedere caso per caso operazioni necessarie al momento della chiusura dell'applicazione (come la chiusura di qualcosa di aperto, la ripulitura, etc.). Se l'applet verrà poi ricaricata, il suo stato avrà perduto ogni traccia della sua precedente esecuzione.

Prima di questo metodo è eseguito sempre il metodo `stop()`.

4.8.4.5 Disegno e metodo `paint()`

Il metodo **`paint(Graphics)`** viene richiamato automaticamente dal browser ogni volta che esso deve disegnare l'interfaccia grafica di un'applet nell'area ad essa riservata sulla pagina HTML per esempio all'avvio dell'applet, ma esso viene attivato anche ogni volta che una parte dell'area dell'applet, dopo essere stata coperta da un'altra finestra sullo schermo o essere uscita dalla parte visibile della pagina HTML per scrolling, viene scoperta di nuovo e rivisualizzata e deve quindi essere ridisegnata, ma anche semplicemente quando la finestra del browser contenente l'applet viene spostata sullo schermo. È solo in questo metodo che si può svolgere ogni operazione grafica e quindi quasi sempre il metodo deve essere (ri)definito.

Ma il metodo `paint()` può essere richiamato, oltre che automaticamente dal browser in seguito agli eventi detti, anche esplicitamente dal programma con il metodo **`repaint()`**, che, come è detto nel paragrafo seguente, ha l'effetto di richiamare il `paint())` per ridisegnare completamente l'area dell'applet.

Al metodo `paint()` il sistema di runtime passa come argomento un oggetto di tipo `Graphics` (package `java.awt`) che rappresenta lo strumento di disegno (è quello che si chiama un contesto grafico e contiene tutte le informazioni grafiche di disegno dell'applet), che è un oggetto creato dal browser e messo da questo a disposizione dell'applet cui lo passa nel metodo `paint()`, nel quale l'applet lo utilizza.

Essendo il metodo `paint()` responsabile del ridisegno dell'applet ogni volta che essa deve riapparire sullo schermo con tutto il suo contenuto grafico, deve comprendere tutte le informazioni grafiche per ricreare il suo disegno completo sullo schermo, quindi, via via che un disegno si accresce col tempo, bisogna conservare tutta la storia grafica nel metodo. E inoltre, siccome esiste questo solo metodo per ridisegnare, quando accade che il disegno deve essere diverso a seconda delle circostanze, ci sarà bisogno di avere dei *flag* definiti con variabili di istanza per fare un disegno piuttosto che un altro.

Esempio:#

4.8.4.6 Metodo **update()**

Il metodo **update()** viene richiamato dall'AWT in risposta a un `repaint()`. Esso nella versione della superclasse `Component` ripulisce prima il componente riempendolo con il colore di fondo e poi chiama il `paint()` per ridisegnare completamente il componente (vedi paragrafo 7.4.13.2). Quando questo metodo è ridefinito nella classe facendogli contenere il solo metodo `paint()` si evita che il `repaint()` cancelli prima lo schermo e il `paint()` disegna solo ciò che è cambiato, lasciando inalterato il resto della grafica preesistente. Esso viene quindi usato per evitare lo sfarfallio delle immagini (vedi paragrafo 7.4.13.3).

4.8.4.7 Altri metodi della classe **Applet**

Il metodo **getAppletContext()**, che restituisce un'istanza della classe `AppletContext`, permette ad un'applet di accedere a certe funzionalità del browser per fornire informazioni all'utente.

Il metodo **showStatus()** della classe `Applet` permette ad un'applet di scrivere messaggi sulla barra di stato del browser.

```
getAppletContext.showStatus(stringaMessaggio);
```

Ma inviare un messaggio all'utente può essere ottenuto anche in altro modo, per esempio con un `Label` o una `Dialog`.

Il metodo **getAppletInfo()** permette di dare all'applet delle informazioni che possono poi essere utilizzate dall'utente dell'applet, per esmpio dal browser.

```
public String getAppletInfo()
   return ...
```

Il metodo **getDocumentBase()** della classe `Applet` restituisce l'URL (come istanza della classe `URL`) del documento sul quale è presentata l'applet.

4.9 Alcune note sulla grafica di un'applet

Se si vuole disegnare un rettangolo sul bordo dell'area dell'applet come cornice, siccome l'origine delle coordinate della grafica entro l'area dell'applet coincide con il vertice del rettangolo in alto a sinistra sullo schermo e i valori positivi di x vanno verso destra e quelli y verso il basso, si deve scrivere l'istruzione seguente:

```
g.drawRect(0, 0, size().width - 1, size().height-1);
```

Se si disegna una parte di un elemento grafico fuori dell'area dell'applet il disegno sarà semplicemente senza effetto per la parte esterna. Vedremo al paragrafo 7.4.13.5 un caso in cui si disegnano delle scritte animate proprio a partire da posizioni esterne all'area dell'applet.

4.10 Caricamento di immagini in un'applet

Per caricare delle immagini la classe `Applet` dispone del metodo `getImage()`, ma bisogna sapere che questo metodo non carica effettivamente le immagini, anche se restituisce subito un oggetto di tipo `Image` che punta all'immagine reale, perchè il caricamento avviene solo quando l'immagine deve essere specificamente utilizzata, generalmente alla richiesta di un `drawImage()`. L'intento (ragione) di questo è di non caricare la memoria prima che sia veramente necessario. Ma per questo si può verificare un ritardo al momento della presentazione dell'immagine sullo schermo nell'area prevista, che per qualche tempo resterà completamente o parzialmente vuota finchè l'immagine non è completamente disponibile. Per prevenire quest'inconveniente si può ricorrere alla classe `MediaTracker` caricando preventivamente l'immagine su un oggetto di questa classe mediante il metodo `addImage()`: con quest'operazione la classe viene caricata realmente in memoria e resa così disponibile ad una sua presentazione immediata sullo schermo.

Ecco le istruzioni:

```
Image img = getImage(url, nome);
MediaTracker mt = new MediaTracker(this);
mt.addImage(img, 0);  // 0 e un numero d'identificazione dell'immagine
      try {
mt.waitForID(0);                      // attende la fine del caricamento
      } catch(InterruptedException e) { }
```

Nel caso di un'animazione costituita di più immagini si avrà a che fare con un array di immagini `img[n]` e l'indice `0` sarà `n`.

Se le immagini risiedono nella directory del documento HTML dell'applet l'argomento `url` di `getImage()` può essere posto come `getDocumentBase()`.

4.11 Interfaccia `AppletContext`

L'interfaccia **`AppletContext`** si riferisce al "contesto" dell'applet e cioè all'ambiente (browser o appletviewer) che ospita l'applet stessa, e in particolare al documento contenente l'applet e alle altre applet eventualmente contemporaneamente presenti nello stesso documento.

Metodi:

getApplet(String)

restituisce la referenza dell'applet (oggetto di classe `Applet`) avente un determinato nome che fosse presente nelllo stesso documento su cui si presenta l'applet.

getApplets()

restituisce sotto forma di `Enumeration` l'insieme di tutte le applet contemporaneamente presenti nello stesso documento dell'applet.

getAudioClip(URL)

crea un oggetto di tipo `AudioClip` (interfaccia) che fa riferimento ad un file audio-clip presente in un determinato URL.

getImage(URL)

restituisce un oggetto di tipo `Image` che fa riferimento ad un file immagine presente in un determinato `URL` che potrà essere riprodotto sullo schermo. L'oggetto `Image` è sempre creato immediatamente anche se l'immagine non fosse disponibile perchè l'immagine sarà realmente caricata soltanto al momento in cui dovrà essere riprodotta. Il disegno dell'immagine sullo schermo è fatto progressivamente via via che le varie parti di essa sono caricati.

showDocument(URL)

rimpiazza la pagina Web presente dell'applet con quella corrispondente ad un determinato `URL`.

showDocument(URL, String)

rimpiazza la pagina Web presente dell'applet con quella corrispondente ad un determinato `URL`, con la possibilità in più di precisare mrdiante un secondo parametro di ripo String la posizione della nuova finestra secondo la tabella seguente.

"_self"	la nuova finestra è presentata in quella presente
"_parent"	la nuova finestra è presentata in quella parent
"_top"	la nuova finestra è presentata in quella topmost
"_blank"	la nuova finestra è presentata in una nuova top-level senza nome
nome	la nuova finestra è presentata in una nuova top-level chiamata *nome*

showStatus(String)

richiede al browser di mostrare una data stringa nella "status window", che è quella finestra nella quale il browser comunica messaggi all'utente (per intenderci è quella stessa in cui compaiono le scritte prodotte dall'applet con il metodo `System.out.println()`).

Tutti questi metodi vengono usati all'interno di un'applet con istruzioni del tipo:

```
getAppletContext().getApplet("seconda");
```

in quanto si applicano all'interfaccia dell'applet stessa, che si ottiene con il suo metodo `getAppletContext()` che determina l'`AppletContext` dell'applet.

Facciamo un esempio in cui sono presenti sulla stessa pagina due applet, una chiamata "prima" da cui si invocano i diversi metodi, e l'altra chiamata "seconda":

```
// D13ContextApplet (F.Spagna) Metodi dell'AppletContext

class D13ContextApplet extends java.applet.Applet {
   public void init() {
      getAppletContext().getApplet("seconda");
      getAppletContext().getApplets();
         try {
      getAppletContext().getAudioClip(new URL("http://spagna"));
      getAppletContext().getImage(new URL("http://spagna"));
      getAppletContext().showDocument(new URL("http://spagna"));
      getAppletContext().showDocument(new URL("http://spagna"));
      getAppletContext().showDocument(new URL("http://spagna"));
      getAppletContext().showDocument(new URL("http://spagna"));
      getAppletContext().showDocument(new URL("http://spagna"));
      getAppletContext().showDocument(new URL("http://spagna"), "nuova
finestra");
         } catch (MalformedURLException e){}
   }
}
```

4.12 File JAR

Il JAR (Java ARchive) è un formato di file (con estensione `.jar`) con compressione di tipo ZIP che permette di riunire in un unico file diversi altri file, ma con qualcosa in più, e precisamente ha un file manifesto e la possibilità di avere file di firma. Esso è indipendente dalla piattaforma ed è scritto in Java. Se usato per le applet un file di questo tipo può raccogliere il bytecode dell'applet (uno o più file .class) e tutti i file di risorse che essa utilizza, come immagini e suoni: in questo modo, avendosi un'unica transazione HTTP ed essendo i file compressi, si riduce grandemente il tempo necessario per il download. Prima dell'esecuzione dell'applet il file viene prima scaricato dalla rete e quindi separato nei suoi file componenti ed è tra questi che l'applet va a cercare prioritariamente i file di cui ha bisogno. I file JAR permettono in più la firma digitale dell'autore per autentificare l'origine dell'applet.

I principali comandi per creare e maneggiare i file JAR mediante il programma eseguibile chiamato **`jar`**, simili a quelli del programma Unix `tar`, sono presentati nella tabella seguente.

per creare un file JAR:	`jar cvf nomeFileJar listaDiFile`
per avere la lista del contenuto di un file JAR:	`jar tvf nomeFileJar`
per estrarre l'intero contenuto di un file JAR:	`jar xvf nomeFileJar`
per estrarre uno specifico file da un file JAR:	`jar xvf nomeFileJar fileDaEstrarre`

Per specificare in un HTML l'uso di un file JAR con un'applet:

```
<applet code=AppletClassName.class
        archive="nomeFileJar.jar"
```

```
            width=width height=height>
</applet>
```

Continua#

SOMMARIO